CROSSTALK

SOFTWARE TOOLBOX   PROCESSES

ANTI-TAMPER TECHNOLOGY

COLLABORATION

SAFETY ANALYSIS

MEASUREMENT

CONFIGURATION MANAGEMENT

TEST AUTOMATION

SOFTWARE TOOLBOX

TECHNIQUES

SOFTWARE PROGRAMS

PROCESSES

Report Documentation Page

1. REPORT DATE **NOV 2004**	2. REPORT TYPE	3. DATES COVERED **00-00-2004 to 00-00-2004**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
CrossTalk. The Journal of Defense Software Engineering. Volume 17, Number 11, November 2004	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **517 SMXS MXDEA,6022 Fir Ave,Hill AFB,UT,84056-5820**	8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**32**	

Software Toolbox

Departments

ON THE COVER
Cover Design by Kent Bingham.

Correction: Author David Schaar's article byline in September's CROSSTALK listed the wrong company affiliation. Schaar works for Booz Allen Hamilton. We apologize for any inconvenience this may have caused.

What's in Your Toolbox?

If you were to start a discussion on software tools, most people's initial frame of reference would probably be tools such as modeling languages, compilers, word processors, project management tools, etc. While these are all important tools, I hope that this issue of CROSSTALK will expand the frame of reference for our readers. When considering software tools that may help with software development and acquisition, projects will realize more benefit if the project team expands its consideration of tools to include helpful processes and techniques in addition to software products such as those listed above.

Over the years, CROSSTALK has shared many tools that apply to most aspects of software development and acquisition; some examples that readily come to mind include updating legacy code, working with people, information security, architectures, and processes such as those promoted in the Capability Maturity Model® (CMM®) Integration and ISO 9000. These are all great tools for improving the quality of software projects, the efficiency in developing software, and the ability to accurately predict the cost and schedule of delivery. The CROSSTALK staff developed this special issue to highlight the idea that tools for developing software are more than just software products. Some of the most useful software tools are the ones most often neglected by software developers, yet much has been expended over the past several years to educate developers (and now acquirers) about these tools and their benefits.

We begin this issue of CROSSTALK with an article from Dr. Alistair Cockburn that truly stresses this point. In *What the Agile Toolbox Contains*, Cockburn discusses numerous tools from all angles of this discussion. If you're not involved with agile software development, you'll see many of these tools apply to other development methods as well. I recommend this article for everyone.

In *A Revolutionary Use of COTS in a Submarine Sonar System*, Capt. Gib Kerr and Robert W. Miller share the success they have achieved thanks to the use of commercial off-the-shelf (COTS) software. As discussed, this effort was not without its drawbacks, but the benefits outweighed the problems.

Next, Dr. Mikhail J. Atallah, Eric D. Bryant, and Dr. Martin R. Stytz discuss various approaches to anti-tamper technologies in *A Survey of Anti-Tamper Technologies*. This discussion emcompasses introductory descriptions of recommended technologies, including their benefits and their drawbacks.

Safety critical software presents additional challenges for the developers. In *Safety Analysis as a Software Tool*, Blair T. Whatcott discusses the basic steps for safety analysis and reminds the readers that safety analysis must be performed at the system level, since many hazards exist at interfaces between system components.

In *Three Essential Tools for Stable Development*, Andy Hunt and Dave Thomas share their experience that configuration management, unit testing, and automation are key to mitigating a majority of the common problems experienced by software developers.

We conclude this issue with a high-level discussion on making measures more useful with David B. Putman's *Your Quality Data Is Talking – Are You Listening?* Putman was one of the key people who helped Hill Air Force Base's Software Engineering Division receive a Level 5 rating on the CMM. In this article, he discusses some of the thought processes that helped so much with the measurement efforts.

You might notice that there are no supporting sections this month. The reason for this is that the CROSSTALK staff believes all of the ideas discussed in these articles should be considered useful tools that support software development. We hope CROSSTALK is also in your software toolbox.

Elizabeth Starrett

Elizabeth Starrett
Associate Publisher

What the Agile Toolbox Contains

Dr. Alistair Cockburn
Humans and Technology

The agile development community is noted for scorning Computer-Aided Software Engineering modeling and Gantt project scheduling tools (among others), but what has it replaced them with? Conducting a survey of agile teams for tools they say help produce better software quicker, this author found they used a cross-disciplinary set of mental, social, environmental, mechanical, and process tools, in addition to a carefully selected set of software-based tools. This list of tools can help your organization prepare for the tools — human resource, facilities, software, and non-software — that will be requested and used by the team starting to adopt the agile approach.

The word *tool* usually brings to mind a physical or software device. However, agile software development teams have removed much of the usually mentioned hi-tech development tools from their repertoire. Thus, in conducting a survey of tools agile teams say produce better software sooner, I had to be more general in considering what might be regarded as a *tool* when asking, "What does the agile toolbox contain?"

The set of tools that agile development teams consider part of their toolbox is very broad, ranging in purpose across *hiring, collaborating, communicating, managing, developing,* etc. Their tools also range in form across *environmental* (such as office layout), *social, physical, process, thinking,* and *computer-based.*

For the survey, I seeded a discussion about tools and posted requests for input to four agile development discussion groups [1, 2, 3, 4]. Originally, I intended to describe a few of the more unusual items on the resulting list. However, the toolset that arrived back was so interesting when considered as a whole that I chose to show it in its entirety.

When people are deciding whether to use an agile development approach on an upcoming project, they can work through this list together, considering the implications of each item on their budget and work habits. Then it will not be such a surprise when the team starts to rearrange the cubicles and request different furniture, post bits of paper all over the wall, or ask to have job applicants co-program with them for a morning.

This article is arranged in the following sections:
- A brief description of agile development with references for further reading and a short description of terms that will show up in the tool lists.
- The tools grouped by the purpose they support.
- The tools itemized by form.
- Reflection on this list as a whole.

Agile Development Acronyms and Key Words

Generally speaking, teams using the agile development approach focus strongly on collaboration and rapid feedback from running code.

> *"Although physical proximity, whiteboards, poster sheets, index cards, and sticky notes are still the dominant tools used in collaboration, people started finding and inventing online collaboration tools as agile development moved into distributed development."*

Collaboration is expected not only within the development team but also across organizational boundaries, with expert users and project sponsors. Collaboration involves group workshop techniques for project planning, requirements gathering and design, and programming in pairs or in close proximity such as in a war-room setting.

Collaboration also involves using *infor-mation radiators* [5] – large displays showing up-to-date information placed in public for people to see whenever they pass by. Information radiators are used in workshops, in the war-room setting, and between continents to keep people in sync on their goal and their state.

Rapid feedback is based on running, tested, integrated system features, or RTF [6]. The project plan is constructed, and progress is measured in terms of the steadily increasing set of integrated features. The team seeks early and frequent integration of features to get feedback about the team, the process it is using, and how the requirements fit the actual needs of the user base.

Attending to collaboration and feedback through RTF drives the selection of many of the tools listed in this article. The agile team cares that the following occurs:
- The right *roles* are established for the team.
- The people who show up *fit* with the rest of the team.
- The people develop particular *skills.*
- The *environment* is effective to the development task.
- They use selected *process* elements.
- *Collaboration* and *communication* are facilitated.
- The *mechanical, hardware,* and *software* tools used are easy to use, see, and update; are effective; and support the agile approach.

Teams debate items in all of these categories and will feel endangered or strengthened when the various items are removed or included. It is on this basis that I consider such a broad range of items *tools.*

There is not space here to undertake a longer description of agile development; it has been heavily described in books and articles. Perhaps the best introduction to the thinking and practices involved is found in "Agile Software Development

Ecosystems" [7] and the articles "The New Methodology" [8], "The Business of Innovation" [9], and "The People Factor" [10]. The AgileAlliance [11] and the Agile Project Management Group [12] offer much more information.

Listed below are some terms that may not be familiar to the reader:

- **Dynamic System Development Method (DSDM):** A founding agile methodology created in the United Kingdom in the late 1990s [13].
- **Scrum:** A founding agile methodology created in the mid-1990s [14].
- **Scrum master:** In the Scrum methodology, a form of team leader who specializes in getting people to talk together and in removing obstacles to progress.
- **Gold cards:** A token allowing a developer to work on something other than scheduled features.
- **Class-Responsibility-Collaborator (CRC) cards:** An object-oriented design technique in which designers write class names on index cards and role-play the design with the cards [15].
- **Java 2 Enterprise Edition (J2EE):** A widely used component library marketed by Sun Microsystems.
- **Unified Modeling Language (UML):** A widely used graphical design documentation notation.

If there are other terms you find unfamiliar, a quick Web search is sure to turn up descriptions and discussions of them.

Tools by Purpose

Included here are entries only for hiring, collaboration, communication, and management purposes. The entries for other activities should be fairly obvious when reading the list grouped by form later in this article.

Hiring

To hire the appropriate people for the team, you must first identify the roles needed and the people to fit those roles. *Process* and *social* tools are used here.

To avoid the standard mistakes in hiring, the tool most often used is a few hours of pair programming with the team members. Teams report being able to tell a lot more about how an applicant thinks, designs, communicates, and fits with the team from this experience. Even without pair programming, interviewers focus on discovering not only an applicant's technical abilities, but also their personal fit with the organization.

Two new roles or skills are sought: facilitators and coaches. In the late 1990s,

the founders of the DSDM felt so strongly that their project teams needed proper facilitation expertise that they helped develop an internationally recognized facilitator training and certification program [16]. An increasing number of software people are becoming certified public facilitators, and more are taking basic facilitator courses.

Coach (from eXtreme Programming [XP]) and *scrum master* (from Scrum) are job titles designed to change the power relationship and interaction dynamics from the traditional team lead or project manager. The coach or scrum master is a lead person whose job typically is to keep desired practices in place and remove obstacles for the group, but not to create schedules for the developers or construct their end-of-year performance reviews. Therefore, the group perceives them as a *leading colleague* rather than a *boss*.

Collaboration

Although physical proximity, whiteboards, poster sheets, index cards, and sticky notes are still the dominant tools used in collab-

> **"Whether collocated or distributed, the two prevalent process tools for collaboration include workshops and short daily status meetings."**

oration, people started finding and inventing online collaboration tools as agile development moved into distributed development. These tools will be listed separately in the computer-based category. They generally include WikiWiki and thread-based discussion group technologies, instant messaging technologies with group and recording variants, and distributed brainstorming technologies.

Whether collocated or distributed, the two prevalent process tools for collaboration include workshops and short daily status meetings. Workshops are used to gather requirements, understand usage patterns, plan the project, and design the software. To support the workshops, specific office facilities are required, including group work areas with lots of wall space, speakerphones, and videoconferencing.

Communication

Active and passive communication

remains a dominant trait of agile development, whether the team is collocated or distributed.

Active communication involves two or more people working on the same task, whether at a whiteboard, sitting side-by-side looking at the same screen, or using shared workspace technology to look at the same screen from different sites.

Passive communication involves information radiators. These are most often on paper or whiteboard. When the information changes on a minute-by-minute basis, information radiators are sometimes driven online. Information radiators include the following:

- A flat monitor hung over the cubicle wall [17].
- A real traffic light hung in the development area that is controlled by an automated build machine [18].
- An ambient orb reporting the same as the traffic light, but using a nationally broadcast signal so teams in all locations can see the same information [19, 20].
- The build status maintained on a Web page so the developers can see what happened to the code they just entered.

Management

Agile teams have replaced Gantt charts with earned value and burn-down charts [21], graphs of tests created versus passed, and similar charts. To report these to upper management, collocated teams still like the effects of posters taped to the wall or spreadsheet graphs. A fresh set of online project management tools is entering the market, including Rally, VersionOne, and XPlanner.

Whether online or on paper, these tools report status with respect to RTF, not planning, design, or documentation tasks.

Tools by Form

Here are the tools clustered by their form: environmental, social, physical, process, mental, and computer-based.

Environmental

You are likely to find that the agile team will either request a different office layout or will simply rearrange their given space to enhance collaboration. The following are common desires:

- Common design and programming areas.
- Lots of wall space for posting information radiators.
- Convex or straight desks so people can cluster around the monitor.
- A common couch area with a white-

board (recording type, preferably).
- Kitchen, for social discussions during breaks.

Social

The top social tools are collocating teams and attacking problems in workshop sessions. Other social tools revolve around increasing the tolerance or amicability of people toward each other, giving them a chance to alternate high-pressure work with decompression periods, and allowing them to feel good about their work and their contributions. The following are desired social tools:
- Social roles such as coach, facilitator, and scrum master.
- Collocated teams (for fast communication and also the ability to learn about each other).
- Personal interaction (within and across specialties).
- Facilitated workshop sessions.
- Daily stand-up status meetings.
- Retrospectives and reflection activities.
- Assisted learning provided by lunch-and-learn sessions, pair programming sessions, and having a coach on the project.
- Pair programming (to provide peer pressure).
- A shared kitchen.
- Toys (to allow humor and reduce stress).
- Celebrations of success and acknowledgment of defeat.
- Gold cards issued at an established rate (to allow programmers to investigate other technical topics for a day or two).
- Off-work get togethers (typically a Friday evening visit to a nearby pub, wine-and-cheese party, even volleyball, foosball, or Doom competitions).
- Posting information radiators in unusual places to attract attention (the most unique I have seen is the number of open defects being posted in the bathroom [22].)

Physical Devices

The best physical devices augment individual thinking, group thinking, and social interaction. The following are some of the preferred ones:
- Index cards and Post-it notes (in any gathering of agile developers, someone is likely to have a pack of index cards with them).
- Butcher paper lining walls and halls.
- Whiteboards (standard or moveable, printing, recording, or with a camera).
- Poster sheets (plain paper, 3M sticky, or plastic cling sheets).

Process

Preferred process tools include short, time-boxed iterations, frequent integration, and frequent delivery. Next to these are workshops for various purposes. Some of the tools are both process and social in form, so I risk listing them twice. They are as follows:
- Project planning jam session (XP's planning game [23], Crystal Clear's blitz planning [17], or Scrum's sprint planning).
- Requirements workshop.
- Group design workshop.
- Reflection or retrospective workshop.
- Pair programming session.
- Refactoring code.
- Growing the system (creating a very small but functional implementation, adding both infrastructure and functionality).
- Time boxing.
- Spike prototyping (throwaway prototyping lasting not more than a day or two).
- Early integration.
- Frequent delivery.
- Programmers writing unit tests.
- Customer writing acceptance tests.
- Tracking by earned value, burn-down, or backlog.

Thinking

Agile developers may or may not model the domain with UML, but they do have tools for helping them decide what and how to code, starting with using the brain. Thinking tools include the following:
- Brain-engaged common sense [24].
- Test-first design (assertion-driven design).
- CRC cards.
- KISS (keep it simple, stupid).
- Once-and-only-once code (do not repeat yourself in your code).

Computer-Based

Jeff Patton writes:

> Of course, agile developers have a long history of tool building – I think that started with chimpanzees using sticks to get bugs out of stumps. Today we use xDoclet to generate J2EE interfaces and classes, which is a lot like getting bugs out of stumps [25].

There are enough entries in this list that I need to group them by purpose. Obviously, I am not attempting a full listing of tool vendors, so I name only one or two sample entries of available online tools where that is relevant. I apologize to the other tool suppliers.

Computer-Based Tools By Purpose

Communication/Collaboration Tools

Here are communication or collaboration tools that require software:
- Group discussion technologies such as WikiWiki, Yahoo! eGroups, Lotus Notes, Starteam, NetMeeting, WebEx, phpBB, and blogs.
- Instant messaging, including group messaging, messaging with drawing, and messaging with discussion thread management. Examples include Yahoo! Messenger with Doodle Imvironment (so people can draw at each other as well as talk), Jabber, AIM, GAIM (group chat), Engage Thoughtware (thread management), and Trillian.
- Collaboration software packages such as Marratech, Raindance, Sparrow, Flywheel, Thoughtware, and Borland's Caliber.
- Video projectors for group coding, learning, and discussion sessions.

Documenting Tools

There is an overlap between collaboration tools and documentation tools. Increasingly, teams look for easy ways to put the results of a group workshop into archive format. Often that involves a camera, but sometimes it means using an online tool during collaboration, including the following:
- Recording whiteboards; scanners; and archiving message, discussion, and collaboration tools (the output is simply put or linked into the documentation).
- Generic drawing tools, PowerPoint, Visio, Dia, and ArgoUML (free) replace expensive computer-aided software engineering packages.

Project Tracking Tools

These are online alternatives to poster sheets posted on the wall, and are particularly useful for distributed teams and for projects whose requirements or plans change multiple times per week. They are as follows:
- Spreadsheets (used to hold project plan and status, and derive tracking graphs).
- Software for tracking the project against stories and tasks comes from XPlanner (free), Rally Software's Agile Release Management, Borland's CaliberRM, and VersionOne.

Designing-Programming Tools

Here are the essential tools requested by agile programming teams:

- Configuration management/version control (Concurrent Version System or your favorite).
- Automated unit test harness such as JUnit or any of the xUnit family.
- Automated acceptance test harness such as Fit or FitNesse.
- Automated build system, preferably a continuous build system such as augmented Another Neat Tool or CruiseControl.
- Refactoring development environment (safe refactoring built in) such as Intellij's IDEA, Eclipse, or ReSharper.
- Performance profiling tool such as Jmeter, Jprofiler, or Jprobe.
- Laptops on a wireless network for programming anywhere.

Other Resources

The Internet contains many discussions of social, process, physical, and computer-based tools for agile development. Ken Boucher has created the Web site <www.fairlygoodpractices.com> for collecting a number of social and process tool descriptions.

To discover your own set, simply hold a workshop with the people on your team and ask them what mental, social, environmental, and physical devices help them in their work. My experience is that they will be glad to share, and you will end up with an impressive list of your own.

Reflection on the Lists

I was surprised at the breadth of tools requested by agile teams, by how far back into the hiring cycle these tools extend, and by the number and importance of the social tools. I was surprised at how far the industry has come in supporting distributed teams with distance collaboration and automated build systems.

As I wrote in [5], understanding passes from person to person more rapidly when they are standing next to each other, as when they are discussing at a whiteboard. Agile teams stress using tools that permit the rapid flow of understanding. Some of those tools are social, starting even at the hiring stage. Some tools are technological, helping distributed teams simulate being physically present. Many tools are physical, allowing people to manipulate them in workshops.

If collaboration is one leg that agile development stands on, the other is rapid feedback from running code. Configuration management, automated testing, refactoring, and performance profiling tools are the dominant entries here. As Michael Vizdos reminds us, do not forget to keep brain and common sense engaged [24].◆

References

1. Yahoo Groups. eXtreme Programming <http://groups.yahoo.com/group/extremeprogramming/message/93430>.
2. Yahoo Groups. Scrum Users <http://groups.yahoo.com/group/scrumdevelopment/message/3652>.
3. Yahoo Finance Groups. Agile Project Management <http://finance.groups.yahoo.com/group/agileprojectmanagement/message/2424>.
4. Yahoo Groups. Salt Lake Agile Software Development <http://groups.yahoo.com/group/sl-agile/message/646>.
5. Cockburn, A. Agile Software Development. Addison-Wesley, 2002.
6. Jeffries, R. "A Metric Leading to Agility." XP Magazine 14 June 2004 <www.xprogramming.com/xpmag/jatRtsMetric.htm>.
7. Highsmith, J. Agile Software Development Ecosystems. Addison-Wesley, 2002.
8. Fowler, M. "The New Methodology." Apr. 2003 <www.martinfowler.com/articles/newMethodology.html>.
9. Highsmith, J., and A. Cockburn. "Agile Development 1: The Business of Innovation" <http://alistair.cockburn.us/crystal/articles/asdboi/asd1businessofinnovation.htm>.
10. Cockburn, A., and J. Highsmith. "Agile Development 2: The People Factor." <http://alistair.cockburn.us/crystal/articles/asdpf/asd2peoplefactor.htm>.
11. Agile Alliance <http://agilealliance.org>.
12. Agile Project Management <http://agileprojectmgt.com>.
13. DSDM Consortium <http://www.dsdm.org>.
14. Control Chaos.com. ADM, Inc. <http://controlchaos.com>.
15. Beck, K., and W.A. Cunningham. "A Laboratory for Teaching Object-Oriented Thinking." ACM SIGPLAN 24.10 (1989): 1-7.
16. Airth, Alan. GlobalFN. Personal communication to the author. 29 June 2004.
17. Cockburn, A. Crystal Clear. Addison-Wesley, 2004.
18. Freeman-Benson, B., and A. Borning. "YP and Urban Simulation: Applying an Agile Programming Methodology in a Politically Tempestuous Domain." Seattle, WA: University of Washington, 2003 <http://agiledevelopmentconference.com/2003/schedule/researchpapers.html#P1>.
19. Ambient Devices <www.ambientdevices.com>.
20. Savoia, Alberto. "eXtreme Feedback for Software Development." Agitar Software, Inc., 2003 <www.developertesting.com/managed_developer_testing/000036.html>.
21. Cockburn, A. "Earned-Value and Burn Charts." Humans and Technology, 22 June 2004. Extracted from Crystal Clear, Addison-Wesley, 2004 <http://alistair.cockburn.us/crystal/articles/evabc/earnedvalueandburncharts.htm>.
22. Developertesting.com <http://www.developertesting.com/images/entryimages/mdt-extreme-feedback_07_0001.jpg>.
23. Beck, K., and M. Fowler. Planning eXtreme Programming. Addison-Wesley, 2001.
24. Vizdos, Michael. Online Posting. Yahoo Groups. Scrum Users <http://groups.yahoo.com/group/scrumdevelopment/message/3661>.
25. Patton, Jeff. Online Posting. Yahoo Groups. Salt Lake Agile Software Development <http://groups.yahoo.com/group/sl-agile/message/648>.

About the Author

Alistair Cockburn, Ph.D., is an internationally respected expert on object-oriented design, software development methodologies, use cases, and project management. He is the author of two Jolt Productivity award winning books, "Agile Software Development" and "Writing Effective Use Cases," as well as author of "Surviving OO Projects." He was also one of the authors of the "Agile Development Manifesto." Cockburn defined an early agile methodology for the IBM Consulting Group in 1992, served as special advisor to the Central Bank of Norway in 1998, and has worked in companies from Scandinavia to South Africa, North America to China. Internationally, he is known for his seminal work on methodologies and use cases, as well as his lively presentations and interactive workshops. Many of his materials are available online at http://alistair.cockburn.us.

Humans and Technology
1814 Fort Douglas CIR
Salt Lake City, UT 84121
E-mail: acockburn@aol.com

A Revolutionary Use of COTS in a Submarine Sonar System

Capt. Gib Kerr
Program Executive Office Submarines

Robert W. Miller
Anteon Corporation

The AN/BQQ-10(V) Acoustic Rapid Commercial off-the-shelf (COTS) Insertion (A-RCI) submarine sonar system has been repeatedly cited as one of the Department of Defense's premier examples of using COTS technology to provide significantly improved system performance at far lower costs than previously possible. The ability to rapidly and inexpensively upgrade a ship's sonar hardware suite to provide continually increasing sonar performance has helped to restore United States submarine superiority over all potential adversaries. As part of this revolution in RCI, the program has identified several lessons on using COTS hardware and software that can help other programs making the same leap into the COTS world.

By the mid-1990s, the United States Navy's submarine force had lost its once seemingly insurmountable lead in detecting and tracking foreign submarines. The use of improved acoustic quieting measures on foreign submarines as well as the worldwide proliferation of modern diesel-electric submarines had sharply reduced the acoustic advantage that the United States had held since the mid-1950s. In addition, the end of the Cold War brought about a significant reduction in available funding to develop and field the improvements necessary to restore superiority. The operating forces were forced to use carry-on commercial systems in an effort to regain some of the advantage that had been lost. These *black boxes* did provide some help but were not fully integrated with the remainder of the ship's combat system, thereby reducing their effectiveness in maintaining tactical control.

In an effort to restore United States submarine sonar superiority and eliminate the need to bring on temporary equipment to meet mission requirements, the Navy began developing the Acoustic Rapid Commercial off-the-shelf (COTS) Insertion (A-RCI) sonar system, later designated the AN/BQQ-10(V). Knowing that the $1.5 billion development cost and the $90 million shipset cost for a new military specification (MIL-SPEC) system was unaffordable, the A-RCI sonar system was designed from day one to use COTS hardware and software components to provide the most up-to-date and powerful computer processing capability possible. This allowed the use of advanced signal processing algorithms to exploit the much quieter target acoustic signatures now available.

Using these advanced algorithms, the U.S. Navy submarine force has now regained the tactical advantage, and an ongoing technology insertion program means that improvements will continue to

be made. In addition, using COTS components instead of MIL-SPEC hardware brought the development cost down to about $100 million and the shipset cost down to $10 million. Since the A-RCI system was designed to replace the different sonar systems on the various submarine classes with a common system, it also reduced the support infrastructure and made it possible for all submarines to have the most modern and capable sonar system available. Commonality also makes it easier to improve the maintenance and operational skill level, and increase the operational experience of the sailors serving in the fleet. The A-RCI program's experiences in using COTS for a critical military system can be of great benefit for other defense programs making the same leap into the COTS world.

Initial Implementation

The first A-RCI hardware suite consisted of a combination of custom and COTS Versa Module Europa (VME)[1] cards to provide the necessary processing power in the limited space available on a submarine. COTS operating systems and hardware drivers were used to the maximum extent practical to minimize the scope of the required software development effort. However, several limitations with this architecture were soon discovered.

The custom cards were prone to failure and were difficult to program. Although technically a COTS product, the signal processing cards were very specialized, leading to high procurement costs and the use of an operating system with limited peripheral driver support. The implementation of the sonar system also used the COTS hardware and software in non-standard ways (i.e., fibre channel standard networks for interprocessor communications vice disk access, Asynchronous Transfer Mode local area networks) making it more difficult to get vendor support or leverage lessons learned from commer-

cial implementations.

Finally, since the A-RCI program was only a small player in the COTS market, receiving timely vendor support for problems found during integration and test was a hit or miss affair. If the vendor felt we were a valuable customer, we would get good support for correcting noted problems; but more likely, the vendor focused its efforts in fixing problems discovered by its more mainstream customers.

The most important lesson learned from this implementation was that as more mainstream hardware and software components were used, fewer problems were discovered during testing, and the vendor was more likely to fix the problems. This revelation became one of the tenets for the technology insertion process that would soon be implemented.

The Technology Insertion Process

One of the key enablers for both the technology insertion process and using COTS hardware in the A-RCI sonar system is using Multipurpose Transportable Middleware (MTM) to isolate the application code from the underlying hardware and its associated drivers and operating systems. MTM was developed and is still maintained by Digital Systems Resources, now a part of General Dynamics Advanced Information Systems.

MTM is a freely licensed set of software utilities that allows for high-speed data passing between the various application software modules running in the A-RCI sonar system, while isolating the modules from the hardware and network protocols. This isolation allows the hardware and associated drivers to be updated without impacting the large amounts of complex application code. Instead, the impact of the hardware change is limited to the MTM that was designed to easily handle change.

By isolating change from the applica-

tion code, many hours (and dollars) are saved with each hardware technology insertion. Because of MTM's benefits, the A-RCI sonar system's hardware has been successfully upgraded five times in the last seven years to reduce system cost and complexity and improve system-processing performance.

The first two technology insertions to the A-RCI hardware baseline were done to eliminate most of the custom VME cards in the system and to provide improved display performance. Elimination of the custom VME cards reduced system cost, improved system reliability, and made software programming easier and faster. Instead of having to code at an assembly level to discrete hardware components, the code could be written in a high-level language (typically C), and features of the COTS operating system could be used to the maximum extent. Simplifying the coding allowed the programmers to spend more time writing better code and debugging problems instead of dealing with the details of the hardware interface.

The VME signal processors with its associated proprietary operating systems and interfaces continued to be used to meet the processing density requirements. However, the decision was made to migrate the display system from VME to a commercial workstation technology when it became apparent that there would be little vendor support for high performance graphics on VME processor boards.

After a survey of available high-end computer workstations, the decision was made to use the HP J5000 workstation and the HP-UX operating system. The choice of this widely used COTS operating system opened the door for display development using standard Motif and Open Graphics Library software libraries. Using standard libraries and their application programming interfaces have made possible rapid updates to the displays to fix problems and implement fleet-user recommendations. This rapid response to user need has become a hallmark of the A-RCI program.

Starting in 2000, the performance levels of mainstream COTS processors became high enough to consider using them for complex signal processing applications. Since then, the technology insertion process has focused on migrating the remainder of the sonar system to mainstream COTS processors with a mainstream operating system.

Market surveys in 2000 indicated that Intel x86 family processors would increase its domination of the server market and that the Linux operating system would become widely supported by device developers. Based on this research, the signal processing applications were shifted from VME cards to Compaq eight-way Pentium III servers running the Linux operating system. An immediate impact of this decision was a large decrease in system acquisition cost. In addition, shifting to a symmetric multiprocessor (SMP) architecture freed the programmer from having to discretely control each individual processor and allowed focusing on making the application code as robust and reliable as possible. Another benefit of using the open-source Linux operating system was its broad user/developer base to help troubleshoot problems. Linux and the software written to use it are also more familiar to most software programmers, leading to higher productivity.

The 2002 and 2004 technology insertions continued the migration to mainstream COTS hardware and software. The

> *"The most important lesson learned from this implementation was that as more mainstream hardware and software components were used, fewer problems were discovered during testing, and the vendor was more likely to fix the problems."*

signal processing servers were changed from the eight-way SMP servers to less expensive dual processor Intel XEON-based servers running at higher clock speeds. In addition, the display servers were changed to dual processor Intel XEON-based servers to reduce the number of different hardware types/operating systems present in the sonar system. Since both the display and signal processing servers now used a common hardware baseline, software development was easier because data transfer was now simpler (no more byte swapping), and a common set of device drivers could be used for both server types. Just as important, the dual processor architecture maintained the pre-

vious generation's flexibility of not having to individually program each processor.

To the maximum extent possible, the system networks were also migrated to Gigabit Ethernet to stay within best commercial practices and provide the most robust set of hardware and device drivers. However, the scope of change in the 2000 and 2002 technology insertions resulted in significant changes to the system network and cabinet enclosures from the previous generation. Therefore, as part of the technology insertions in 2002 and 2004, a concerted effort was made to make the system network architecture more flexible and to make the cabinet enclosures easier to upgrade during future technology insertions. This effort is succeeding as the cabinet enclosure and cabling system differences between the 2002 and 2004 technology insertions are minimal. Now, when a new processor design is chosen in a future technology insertion, no multi-million dollar cabinet redesign will be required.

Eliminating the need to redo a large portion of the shipboard cabling and change-out cabinets will also ensure that future technology insertions can be done in a standard length port maintenance period of about 35 days for about 20 percent of the cost it had previously taken. Reducing change external to the cabinets is imperative to minimizing the shipboard impact of technology insertion.

The Benefits of COTS

The A-RCI sonar program takes advantage of the many benefits of using COTS hardware and software for military applications. A significant benefit is the ability to use computer systems much closer to commercial state-of-the-art systems than was ever possible with MIL-SPEC systems. This has allowed the use of advanced computation-intensive signal processing algorithms and easy-to-use displays to improve the operator's ability to detect signals of interest. Using COTS processors also makes it much easier to develop, purchase, and install upgrades to the sonar system to keep its performance at the highest possible level.

In addition, using standard rack-mounted server boxes means that ongoing improvements in commercial computers can now be rapidly inserted into the system with minimal changes required. This is similar to the way a business using Hewlett Packard and Dell computers would upgrade its server farm. Importantly, the ongoing technology insertion process eliminates the need to maintain obsolete COTS hardware; instead, when a ship's computer hardware

becomes obsolete and unsupportable, it is replaced with an up-to-date system.

Using COTS hardware components brings the benefit of using COTS operating systems, device drivers, and libraries. This has enabled the system software developers to focus on the applications versus the support software. In addition, mainstream software is better tested and more robust than custom software. Using open-source software such as Linux brings the advantage of a large developer base so that software problems will be resolved in a very timely manner. The large developer base also ensures that any security holes are quickly discovered and corrected and that no malicious code is inserted into the operating system. This is one reason why Linux has a much lower incidence of security breaches than the proprietary Microsoft Windows operating systems. Although COTS operating systems, device drivers, and libraries are used, the critical application software is still written and maintained by the system developers using secure facilities.

The lower hardware cost and the continuous improvement cycle associated with commercial computer hardware is what allows the A-RCI technology insertion process to succeed. If the cost of hardware components were equivalent to the MIL-SPEC hardware used in the past, the pace of system upgrades would be unaffordable and the Navy would soon be behind the technology curve like it was in the mid-90s. Using a COTS technology insertion process has enabled a 10x increase in system throughput and an 86 percent reduction in hardware cost per billion floating point operations per second in a six-year period. Low hardware cost has also allowed the A-RCI sonar program to purchase system equipment from several vendors, ensuring that a continuous price competition exists.

Because integrating COTS components is within the capability of firms much smaller than the traditional major Department of Defense (DoD) contractors, a much broader business base is also available. Configuration control of the system is maintained by requiring all system equipment vendors to work together in specifying the COTS components.

The Downside of COTS

The downside to using COTS software is the lack of insight into the code details. Since the system contractor does not write the software, the programmers have a much-reduced understanding of the

code than they would have with internally developed software. This could make the development team dependent on the skills of the open-source community to fix any problems noted during integration and testing – an unacceptable situation. This situation is prevented by researching the COTS products selected to verify they are in use by many other developers with similar applications and requirements. This broad user base helps ensure the software is well tested and robust before it is used by the A-RCI system.

A more significant downside is a result of using COTS hardware in a non-office environment. COTS servers are designed for use in well air-conditioned spaces and not the sealed, water-cooled cabinets used on submarines. Cooling the processors has become a significant issue, currently limiting the team's ability to use the full capability of today's processors. In the future, it may not be possible to continue providing increased processing power with each technology insertion unless improved cabinet cooling methods can be implemented.

Process Migration

The benefits of this new COTS business model have so significantly outweighed the disadvantages (primarily with respect to cost and rate at which capability can be added) that the model has been expanded to include the entire non-propulsion electronics suite on the newest class of submarine: the USS Virginia attack submarine. The process has expanded from what was simply a single sonar sensor and processor to a 20-million source lines of code system of systems that includes *all* sensors, ship's navigation, combat/fire control, and ship monitoring functions.

Rapid COTS insertion is also being used to upgrade older submarine classes' combat control systems and is planned for use on undersea weapons. This ability to rapidly insert improved capability in the form of software and hardware has become a hallmark of acquisition reform. Software and hardware solutions that are one-time developments are now implemented in many systems, including those in use on submarines, surface ships, undersea surveillance systems, and aircraft.

Conclusion

The AN/BQQ-10(V) A-RCI sonar system would not be the success it is today without its embrace of COTS hardware and software. The only way to economically take advantage of the advances in

computer processing is to buy from the mainstream market. The less the hardware has to be modified to work in the system, the more rapidly and inexpensively it can be implemented. Moreover, COTS hardware brings with it COTS software. The contractor must learn to live within the limitations of the software and not try to make it incrementally better. Time spent in this manner is time not spent improving the more critical system application software. By picking COTS software that is well used and tested, the contractor can reduce problems observed, but also must accept the loss of total control over the code.

If COTS hardware is used, an ongoing technology insertion program is required to reduce obsolescence issues and maintain the system at its highest capability. A-RCI has successfully implemented five technology insertions and has an ongoing plan to continue with the process. Making the technology insertion process affordable is the MTM, which helps to isolate the complex application code from the underlying hardware and device drivers. A-RCI has shown that it is possible to reap the benefits of COTS computer hardware and software while still meeting all military requirements. It is now up to other DoD programs to make the same leap.

Final Points

Adopting a RCI process is not painless. Overcoming organizational bias, MIL-SPEC thinking, severe skepticism, and the not-invented-here syndromes were tremendous challenges for the A-RCI program to overcome in its early stages. It required an extraordinary culture shift for all stakeholders to achieve what today is almost taken for granted. The combined sense of urgency due to 1) the need to regain technological superiority, and 2) severe budget cuts drove the U.S. Navy's submarine force to the RCI solution. Without those kinds of drivers, no amount of hearing *this is a good idea* will result in other DoD programs adopting RCI processes. RCI is a business decision that requires dedicated believers to succeed and change the *status quo* of systems acquisition, and properly leverage the power and agility of the commercial, non-government business world.

Knowing that RCI may be the only efficient way to quickly regain technological superiority at a reduced cost does not mean that those who manage such programs should always sleep well at night. What have been discussed here are implementations in modern *sensor and*

combat systems. These are systems that warfighters depend on when they are in harm's way to be 100 percent effective. It is this balance between efficiency in cost and effectiveness in war that should keep program managers awake at night with these questions:

• How can we be 100 percent assured that COTS products contain no latent defects that may have deadly consequences when they manifest themselves?
• What represents the necessary and sufficient testing and verification to preclude unacceptable consequences?
• How much do we really want to be dependent on a potentially fickle commercial market for critical systems in our military machines?
• What is the minimum acceptable cost/risk ratio for a critical technology?
• What is the necessary and sufficient amount of discipline required in the process so that capability is rapidly inserted, without undo risk, and without unduly constraining the process?

It is the description, quantification, understanding, and reconciliation of these issues and their risks that must become the main focus and challenge of the program manager's efforts in an RCI program. They certainly have become the focus for the submarine force's A-RCI program.◆

Note

1. VME is a standard developed in 1981 for embedded computer hardware form factor and data transfer protocol.

About the Authors

Capt. Gib Kerr is a mechanical/nuclear/acoustic engineer by training, a nuclear submariner by qualification, a combat systems program manager by assignment, and a systems acquisition professional by choice. He has done a little of everything, including traditional jobs onboard submarines, flag officer's staff, repaired and built submarines, and has been a fleet repair officer (repairing surface ships, submarines, and one helicopter). A consummate team player and trained facilitator, he has led, been a member of, and facilitated numerous Integrated Product Teams and Process Action Teams and facilitated several organizational reengineering and restructuring projects. Kerr's strength is in getting disparate organizations working together, assessing performance and value management to develop and implement sound technical and business solutions. He has been working Navy submarine acquisition programs for the past seven years.

Program Executive
Office Submarines
Attn: PMS 401
614 Sicard ST SE STOP 7013
Washington Navy Yard, D.C.
20376-7013
Phone: (202) 781-1556
Fax: (202) 781-4688
E-mail: kerrgb@navsea.navy.mil

Robert W. Miller is a senior program manager with Anteon Corporation. For the past eight years, he has been providing technical support for the Acoustic Rapid Commercial off-the-shelf Insertion sonar program to the Submarine Acoustic Systems Program Office. Prior to joining Anteon, he served in the United States Navy for 16 years as a submarine line officer and engineering duty officer. He has a Bachelor of Science in electrical engineering from the United States Naval Academy and a Master of Science in electrical engineering from the Naval Postgraduate School.

Anteon Corporation
1100 New Jersey AVE SE STE 200
Washington, D.C. 20003
Phone: (202) 756-7629
Fax: (202) 646-0122
E-mail: rwmiller@anteon.com

A Survey of Anti-Tamper Technologies

Dr. Mikhail J. Atallah, Eric D. Bryant, and Dr. Martin R. Stytz

Arxan Technologies, Inc.

This article surveys the various anti-tamper (AT) technologies used to protect software. The primary objective of AT techniques is to protect critical program information by preventing unauthorized modification and use of software. This protection goal applies to any program that requires protection from unauthorized disclosure or inadvertent transfer of leading-edge technologies and sensitive data or systems. In this article, we review the various approaches to AT techniques, their strengths and weaknesses, their advantages and disadvantages, and briefly discuss a process for developing program protection plans. We also survey the tools that are typically used to circumvent AT protections, and techniques that are commonly used to make these protections more resilient against such attack.

The unauthorized modification and subsequent misuse of software is often referred to as software *cracking*. Usually, cracking requires disabling one or more software features that enforce policies (of access, usage, dissemination, etc.) related to the software. Because there is value and/or notoriety to be gained by accessing valuable software capabilities, cracking continues to be common and is a growing problem.

To combat cracking, anti-tamper (AT) technologies have been developed to protect valuable software. Both hardware and software AT technologies aim to make software more resistant against attack and protect critical program elements. However, before discussing the various AT technologies, we need to know the adversary's goals. What do software crackers hope to achieve? Their purposes vary, and typically include one or more of the following:

- **Gaining unauthorized access**. The attacker's goal is to disable the software access control mechanisms built into the software. After doing so, the attacker can make and distribute illegal copies whose copy protection or usage control mechanisms have been disabled – this is the familiar software piracy problem. If the cracked software provides access to classified data, then the attacker's real goal is not the software itself, but the data that is accessible through the software. The attacker sometimes aims at modifying or *unlocking* specific functionality in the program, e.g., a demo or *export* version of software is often a deliberately degraded version of what is otherwise fully functional software. The attacker then seeks to make it fully functional by re-enabling the missing features.
- **Reverse engineering.** The attacker aims to *understand* enough about the software to steal key routines, to gain access to proprietary intellectual property, or to carry out *code-lifting*, which consists of reusing a crucial part of the code (without necessarily understanding the internals of how it works) in some other software. Good programming practices, while they facilitate software engineering, also tend to simultaneously make it easier to carry out reverse engineering attacks. These attacks are potentially very costly to the original software developer as they allow a competitor (or an enemy) to nullify the developer's competitive advantage by rapidly closing a technology gap through insights gleaned from examining the software.
- **Violating code integrity.** This familiar attack consists of either injecting malicious code (*malware*) into a program, injecting code that is not malevolent but illegally enhances a program's functionality, or otherwise subverting a program so it performs new and unadvertised functions (functions that the owner or user would not approve of). While AT technology is related to anti-virus protection, it has some crucial differences. AT technology is similar to virus protection in that it impedes malware infection of an AT-protected executable. However, AT technology differs from virus protection in that the AT technology's goal is not only to protect the client's software from unauthorized modification by malevolent outsiders (infection by malware written by others), but also to protect the software from modification by an authorized client. In many situations, it is important that only authorized applications execute (e.g., in a taximeter, odometer, or any situation where tampering is feared), using only authorized functionality, and that

only valid data is used.

It should be clear by now that AT technology is not only about anti-piracy, it has an equal and broader aim of policy enforcement. That aim is to enforce the policies of the software publisher about the proper use of the software, even as the software is running in a potentially hostile environment where the user *owns the processor* and is intent on violating those policies.

There is a plethora of AT protection mechanisms. These include encryption wrappers, code obfuscation, guarding, and watermarking/fingerprinting in addition to various hardware techniques. While these techniques are discussed separately for pedagogical purposes, the reader should bear in mind that software is best protected when several protection techniques are used together in a mutually supportive manner. No technique is invulnerable or even clearly superior to the others in all circumstances; therefore, a mix of protection techniques allows the defense to capitalize on the strengths of each technique while also masking the shortfalls of other techniques. In the following paragraphs we present a brief overview of these techniques.

Hardware-Based Protections

The most common hardware approach uses a trusted processor. The trusted, tamper-resistant hardware checks and verifies every piece of hardware and software that exists – or that requests to be run on a computer – starting at the boot-up process [1]. This hardware could guarantee integrity by checking every entity when the machine boots up, and every entity that will be run or used on that machine after it boots up. The hardware could, for example, store all of the keys necessary to verify digital signatures, decrypt licenses, decrypt software before running it, and encrypt messages during

any online protocols it may need to run (e.g., for updates) with another trusted remote entity (such as the software publisher).

Software downloaded onto a machine would be stored in encrypted form on the hard drive and would be decrypted and executed by the hardware, which would also encrypt and decrypt information it sends and receives from its random access memory. The same software or media could be encrypted in a different way for each trusted processor that would execute it because each processor would have a distinctive decryption key. This would put quite a dent in the piracy problem, as disseminating your software or media files to others would not do them much good (because their own hardware would have different keys).

A less drastic protection than using a separate, trusted, hardware computational device also involves hardware, but is more lightweight such as a smart card or physically secure *token*. These lightweight hardware protection techniques usually require that the hardware be present for the software to run, to have certain functionality, to access a media file, etc. Defeating this kind of protection usually requires *working around* the need for the hardware rather than duplicating the hardware. The difficulty of this work-around depends on the role that the tamper-resistant hardware plays in the protection. A device that just outputs a serial number is trivially vulnerable to a *replay attack* (e.g., an attacker replays a valid serial number to the software, without the presence of the hardware device), whereas a smart card that engages in a challenge-response protocol (different data each time) prevents the simple replay attack but is still vulnerable (e.g., to modification of the software interacting with the smart card). A device that decrypts content or that provides some essential feature of a program or media file is even harder to defeat.

Advantages and Drawbacks

The chief advantage of hardware-based protection techniques is that they run on a trusted CPU and can be made arbitrarily complex – hence, difficult to defeat while inflicting minimal computational cost on the protected software once it has been decrypted within the hardware and is running. However, there is a cost to decrypt it in the first place, and also to encrypt everything that goes out to the non-protected part of the system, and then decrypt it when it comes back into the trusted hardware.

In addition, it is generally more diffi-

cult to successfully attack tamper-resistant hardware and make the exploit directly available to others than a software-only protection scheme. This point holds only for a properly designed system. A compromise of hardware that imprudently contains the same secret keys as all other hardware of the same type would lead to widely reproducible exploits.

The advantages of hardware protection also include its capability to enforce such rules as "only approved peripherals can be a part of this computer system," or "only approved (through digital signatures) software and contents are allowed," etc.

Nevertheless, hardware-based protection also has its drawbacks. There is the usual problem of inflexibility: hardware-based protections are more awkward to modify, port, and update than software-

> **"No technique is invulnerable or even clearly superior to the others in all circumstances; therefore, a mix of protection techniques allows the defense to capitalize on the strengths of each technique while also masking the shortfalls of other techniques."**

based ones. They are also less secure than commonly assumed and can be broken; see, e.g., [2]. To date, it has not been demonstrated that hardware protections can scale to grid computing or to small-scale computing. In addition, there is no guarantee that all avenues of attack are closed by hardware protection, and there is a significant cost attached to using hardware protection; the cost is driven mainly by the time needed to assemble, integrate, and test the hardware protection technique.

Additional drawbacks to the hardware protection approach include its expense and general *fragility* to accidents (an electric power surge that *fries* the processor

also renders the hard drive contents unusable because the key that decrypts them is destroyed). The potential implications for censorship are also chilling. Another disadvantage of hardware protection is the boot-up time and the time spent encrypting and decrypting, which makes the approach problematic for low-end machines and embedded systems (unless the whole system lies within tamper-resistant hardware).

Using trusted hardware also incurs many indirect costs as a result of the earlier-mentioned limitations it imposes (e.g., the restriction to only certain *approved* hardware, software, and media creates a barrier to competition that leads to higher prices). Due to the imperfect protection offered by hardware protection, a more robust approach to software security interweaves hardware protection with other protection techniques such as those discussed in the following sections.

The rest of this article discusses the various software-based protection mechanisms. The reader should keep in mind that hardware and software protection techniques are not mutually exclusive. A judicious combination can serve to increase the security of the system more than any of its individual component techniques.

Encryption Wrappers

With encryption wrapper software security, critical portions of the software (or possibly all of it) are encrypted and decrypted dynamically at run-time. The encryption wrapper approach works well against a static attack, and forces the attacker to run the program in order to get an unencrypted image of it. To make the attacker's task harder, at no time during execution is the whole software *in the clear*; code decrypts just before it executes, leaving other parts of the program still encrypted. Therefore, no single *snapshot* of memory can expose the whole decrypted program. Of course, the attacker can take many such snapshots, compare them, and piece together the unencrypted program.

Another avenue of attack is to figure out the various decryption keys that are present in the software. One defensive technique that can be used to delay the attacker is to include defensive mechanisms in the program that deprive the attacker of using run-time attack tools, e.g., anti-debugger, anti-memory dump, and other defensive mechanisms, which make it more difficult for the attacker to run and analyze the program in a synthetic (virtual machine) environment. Yet, a determined attacker can usually defeat

these protections (e.g., through the use of virtual machines that faithfully emulate a PC, including the most rarely used instructions, cache behavior, etc).

Encryption wrappers often use *lightweight* encryption to minimize the computational cost of executing the protected program. The encryption can be advantageously combined with compression: Not only does this result in a smaller amount of storage usage, but it also makes the encryption harder to defeat by cryptanalysis (of course one compresses before encryption, not the other way around).

An encryption wrapper's chief advantage is that it effectively hinders an attacker's ability to statically analyze a program. The attacker is then forced to perform more sophisticated types of dynamic attacks, which can significantly increase the amount of time needed to defeat the protection. The main disadvantage of encryption wrappers is the performance penalty caused by the decryption overhead, and its weakness to memory dumps: before it can run, encryption-protected software must be decrypted, at which point it becomes exposed.

Code Obfuscation

Code obfuscation consists of transforming code so it becomes less intelligible to a human, thus making it not only harder to reverse engineer, but also harder to tamper with. In software that has specific areas where policy checks are made, these areas will be harder to identify and disable after the software has been obfuscated. Obfuscation is usually carried out by inserting or performing obfuscating transformations. It is a requirement that these transformations do not damage a program's functionality, and it must have only a moderate impact on code performance, and on the storage space used on the disk and at run-time (of the two, speed is more important).

The obfuscation must also be resilient to attack, and for this reason it is desirable to maximize the *obscurity* of the obfuscated software. The obfuscating transformations need to be resilient against tools designed to automatically undo them, and to not be easily detectable by statistical analysis of the resulting code (resilience to statistical analysis makes it harder for automatic tools to find the locations where these transformations were applied).

The different types of obfuscation transformations that have been proposed [3] include the following:

- **Layout obfuscation.** This modifies the *physical appearance* of the code, e.g.,

replacing important variables with random strings, removing all formatting (making nested conditional statements harder to read), etc. Such transformations are easy to make but are effective only when combined with other transformation techniques.
- **Data obfuscation.** This obscures the data structures used within a program, e.g., the representation and the methods of using that data, independent data merging (and vice-versa – splitting up data that is dependent), etc. Data obfuscation serves to delay the attacker because data structures contain important information that any attacker needs to comprehend before launching an attack.
- **Control obfuscation.** This manipulates the control flow of a program to make it difficult to discern its original structure, e.g., through merging (or splitting) various fragments of code, reordering expressions, loops, or blocks, etc. It is similar to creating a spurious program that is *entangled* with the original program so as to obscure the important control features of that program.
- **Preventive transformations.** These aim at making it difficult for a de-obfuscation tool to extract the *true* program from the obfuscated version of it. Preventive transformations can be implemented by using what Collberg [4] calls opaque predicates, an example of which is a conditional statement that always evaluates as true, but in a manner that is hard to recognize.

Obfuscation can be done at the source-code level (source-to-source translation) or at the assembly level. Although most obfuscators are of the former kind (source-to-source), assembly level obfuscation is better because it effectively hides the operation of the binary. If the source-code level transformations hide information by adding crude and inefficient ways of doing simple tasks, then the code optimizer in the compiler may undo them. If, on the other hand, the transformations are clever enough to fool the optimizer, then it can fail to properly do its job, and the performance of the resulting code suffers. Low-level obfuscation does not prevent the code optimizer from doing its job, but if done carelessly it runs the risk of producing code that looks so different from the kind produced by the compiler that it inadvertently *flags* the areas where transformations were applied.

Obfuscation transformations are clas-

sified according to several criteria: how much obscurity they add to the program (potency), how difficult they are to break for a de-obfuscator (resilience), and how much computational overhead they add to the obfuscated application (cost). In [4], software complexity metrics are used to formalize the notion of transformation potency and resilience.

The potency of a transformation measures how much more difficult the obfuscated code is to understand for a human than the original code. On the other hand, the resilience of a transformation measures how well it stands up to attack by an automatic de-obfuscator. The resilience measurement takes two factors into account: the programmer effort required to construct the de-obfuscator and the execution time and space required by the de-obfuscator to reduce the potency of the transformation. The best obfuscation is usually achieved by a combination of the above three mentioned transformations. The combination of the three approaches provides a well-balanced mix of highly potent and resilient transformations.

Like all software-only protections, obfuscation can delay – but not prevent – a determined attacker intent on reverse engineering the software. Barak [5] presents a family of functions that are provably impossible to completely and successfully obfuscate. For more information and a discussion of code obfuscation, refer to [3, 4, 6, 7].

Software Watermarking and Fingerprinting

The goal of watermarking is to embed information into software in a manner that makes it hard to remove by an adversary without damaging the software's functionality. The information inserted could be purchaser information, or it could be an integrity check to detect modification, the placing of caption-type information, etc. A watermark need not be stealthy; visible watermarks act as a deterrent (against piracy, for example), but most of the literature has focused on stealthy watermarks. In steganography (the art of concealing the existence of information within seemingly innocuous carriers), the mark is required to be stealthy: its very existence must not be detectable [8].

A specific type of watermarking is fingerprinting, which embeds a unique message in each instance of the software for traitor tracing. This has consequences for the adversary's ability to attack the

watermark: two differently marked copies often make possible a *diff* attack that compares the two differently marked copies and can enable the adversary to create a usable copy that has neither one of the two marks. Thus, in any fingerprinting scheme, it is critical to use techniques that are resilient against such comparison attacks.

A watermark is generally required to be robust (hard to remove). In some situations, however, a fragile watermark is desirable; it is destroyed if even a small alteration is made to the software (e.g., this is useful for making the software tamper-evident).

Software watermarks can be static, i.e., readable without running the software, or could appear only at run-time (preferably in an evanescent form). In either case, reading the watermark usually requires knowing a secret key, without which the watermark remains invisible.

Watermarks may be used for proof of software authorship or ownership, fingerprinting for identifying the source of illegal information/software dissemination, proof of authenticity, tamper-resistant copyright protection, and captioning to provide information about the software. When software watermarks are used for proof of authorship or ownership (culprit-tracing), it is important to use a very resilient scheme. Recall that this is when the watermark contains information about the copyright owner as well as the entity that is licensed to use the software, thus allowing trace-back to the culprit if the item were to be illegally disseminated to others. Breaking the security of such a scheme can enable the attacker to *frame* an innocent victim.

As you can see, while watermarks can demonstrate authorized possession and the fact that software has been pirated, they do not address the reverse engineering or authorized execution issues of the other schemes discussed; therefore, we advocate the development and use of a spectrum of software protection techniques.

Guarding

A guard is code that is injected into the software for the sake of AT protection. A guard must not interfere with the program's basic functionality unless that program is tampered with – it is the tampering that triggers a guard to take action that deviates from normal program behavior. Examples of guard functionality range from tasks as simple as comparing a checksum of a code fragment to its expected value, to repairing code (in case

it was maliciously damaged), to complex and indirect forms of protection through subtle side effects.

The preferred use of the guarding approach consists of injecting into the code to be protected a large number of guards that mutually protect each other as well as the software program in which they now reside. Guards can also be used to good effect in conjunction with hardware-based protection techniques to further ensure that the protected software is only executed in an authorized environment.

The number, types, and stealthiness of guards; the protection topology (*who protects who*); and where the guards are injected in the original code and how they are entangled with it are some of the parameters in the strength of the resulting protection: They are all tunable in a manner that depends on the type of code being protected, the desired level of protection, etc.

Manually installing such a tangled web of protection is impractical (as it must be done every time the software is updated), so it is important that this protection be done in a highly automated fashion using high-level scripts that specify the protection guidelines and parameters. It should be thought of as a part of the compilation process where an *anti-tamper* option results in code that is guarded and tamper-resistant.

The rationale for this approach is that a single (even if elaborate) AT protection scheme for a software application is insufficient because a single defense results in a single point of attack that can be located and compromised. To make self-protection robust, the defense must not rely on a single complex protection technique no matter how effective it might be. Instead, there needs to be a multitude of (possibly simple) protection techniques installed in the program that cooperatively enforce the code's integrity as well as protect the other against tampering.

The guard's response when it detects tampering is flexible and can range from a mild response to the disruption of normal program execution through injection of run-time errors (crashes or even subtle errors in the answers computed); the reaction chosen depends on the software publisher's business model and the expected adversary. Generally, it is better for a guard's reaction to be delayed rather than to occur immediately upon detection so that tracing the reaction back to its true cause is as difficult as possible and consumes a great deal of the attacker's time.

More on guarding can be found in [9].

AT Process

This section explores the various AT guidelines expressed in the "Defense Acquisition Guidebook" [10], and the recommended process for developing a program protection plan. The "Defense Acquisition Guidebook" specifies the AT measures that should be considered for use on any system with critical program information (CPI), developed with allied partners, likely to be sold or provided to United States allies and friendly foreign governments, or likely to fall into enemy hands. The first step in the recommended AT methodology is to develop a program protection plan. The process of developing this plan includes the following:

- Develop a list of critical technologies.
- Develop a threat analysis.
- Develop a list of identified vulnerabilities.
- Develop a preliminary AT requirement.
- Perform an analysis of AT methods that applies to the system, including cost/benefit assessments.
- Provide an explanation of which AT method(s) will be implemented; develop a plan for validating the AT implementation.

The standard approach of validating AT protections is done via *red-teaming*. A red team consists of individuals who are well versed in security methods and their corresponding weaknesses. Their primary objectives are to attempt to defeat the protection, to assess the protection's strengths and weaknesses, and to make recommendations for improvement. While this is an effective method of evaluation, a major problem with red teams is that the validation is done by humans, and may not be totally reliable or repeatable. Furthermore, as the need for AT technologies grows, red teams are becoming increasingly called upon to perform evaluations. The teams are overwhelmed with assignments, significant delays in product evaluations, and release results. To improve this process, there is a clear and present need for automated testing and validation tools. Such tools could be used to define a standard set of metrics and guidelines to evaluate software protections.

Conclusion

This article has surveyed the motivation for using AT technology, the hardware and software AT techniques in use today, and the strengths and weaknesses of AT technologies. We also briefly introduced

the process and documentation used to develop a program protection plan. The motivation for and primary objective of AT technology is to protect CPI by preventing unauthorized modification and use of software. The main software AT techniques include encryption wrappers, code obfuscation, watermarking/fingerprinting, and guarding.

A fundamental challenge faced by software AT technology is that the protected application is running on a host that is not trusted, and thus cannot be assured to be secure. Guards address this shortfall to a degree and in a flexible and extensible manner. However, in light of the need for robust, seamless, comprehensive software defense, using both software and hardware AT solutions in combination often offers an appealing alternative to using them individually (especially if economic considerations are factored in).

At this time, indications are that if strong software AT technology (e.g., in the form of judiciously constructed guards) is added to an application so that it requires the presence of a *lightweight* tamper-resistant hardware device in order to execute properly, the result is a strong yet economical software protection capability.◆

References

1. Arbaugh, W., D. Farber, and J. Smith. <u>A Secure and Reliable Bootstrap Architecture</u>. Proc. of the IEEE Symposium on Security and Privacy, Oakland, CA, 1997.
2. Anderson, R., and M. Kuhn. <u>Tamper Resistance – A Cautionary Note</u>. Proc. of Second Usenix Workshop on Electronic Commerce, Oakland, CA, Nov. 1996: 1-11.
3. Collberg C., and C. Thomborson. "Watermarking, Tamper-Proofing, and Obfuscation Tools for Software Protection." <u>IEEE Transactions on Software Engineering</u> 28.8 (2002): 735-746, 2002.
4. Collberg, C., C. Thomborson, and D. Low. "A Taxonomy of Obfuscating Transformations." Department of Computer Science, University of Auckland, New Zealand, 1997.
5. Barak, B., et al. "On the (Im)possibility of Obfuscating Programs." <u>Electronic Colloquium on Computational Complexity</u>. Report No. 57, 2001.
6. Wang, C., et al. "Software Tamper Resistance: Obstructing Static Analysis of Programs." University of Virginia, Computer Science Technical Report CS-2000-12, Dec. 2000.
7. Wroblewski, G. "General Method of Program Code Obfuscation." Diss. Wroclaw University of Technology, Institute of Engineering Cybernetics, 2002.
8. Johnson, N. "Introduction to Steganography and Steganalysis." Workshop on Statistical and Machine Learning Techniques in Computer Intrusion Detection, Johns Hopkins University, 11-13 June 2002.
9. Chang, H., and M. Atallah. <u>Protecting Software Code By Guards</u>. Proc. of ACM Workshop on Security and Privacy in Digital Rights Management, Philadelphia, PA, Nov. 2001: 160-175.
10. Office of the Secretary of Defense. <u>Interim Defense Acquisition Guidebook</u>. Washington, D.C.: OSD, 30 Oct. 2002 <http://dod5000.dau.mil/DoD5000Interactive/InterimGuidebook.asp>.

Additional Reading

1. Lipton, R.J., S. Rajagopalan, and D.N. Serpanos. "Spy: A Method to Secure Clients for Network Services." <u>IEEE Distributed Computing Systems Workshops</u> 2002: 23-28.
2. Anderson, R., and M. Kuhn. "Low Cost Attacks on Tamper Resistant Devices." 5th International Workshop on Security Protocols, Apr. 1997: 125-136.

About the Authors

Mikhail "Mike" J. Atallah, Ph.D., is a distinguished professor in the Computer Science Department at Purdue University. His main research interests are in information security. A fellow of the Institute of Electrical and Electronics Engineers, Atallah has been both a keynote and invited speaker at many national and international meetings, and a speaker in the Distinguished Colloquium Series of many top computer science departments. He is a co-founder of Arxan Technologies Inc. Atallah has a Master of Science and a doctorate degree from Johns Hopkins.

Arxan Technologies, Inc.
3000 Kent AVE
STE D2-100
West Lafayette, IN 47906
Phone: (765) 494-6017 ext. 54
Fax: (765) 496-3181
E-mail: mja@cs.purdue.edu

Eric D. Bryant is a research engineer for Arxan Technologies, Inc., and is pursuing his doctorate degree at Purdue University. His primary research interests are in information security, reverse engineering, compiler and programming language design, and artificial intelligence. Bryant has a Bachelor of Science in computer science from Purdue University. Arxan's EnforcIT™ product fortifies software applications with complex software guards designed to prevent unauthorized access, reverse engineering, and code lifting. More information can be found at <www.arxan.com>.

Arxan Technologies, Inc.
3000 Kent AVE
STE D2-100
West Lafayette, IN 47906
Phone: (765) 775-1004 ext. 106
Fax: (765) 775-1004
E-mail: ebryant@arxan.com

Martin R. Stytz, Ph.D., is a senior research scientist and engineer for Calculated Insight and formerly for the Air Force Research Laboratory at Wright-Patterson Air Force Base, Ohio. Stytz was a consultant at Arxan Technologies, Inc. at the time this article was written. He is a member of the Institute of Electrical and Electronics Engineers (IEEE) Task Force on Security and Privacy, and is on the editorial board for IEEE's Security and Privacy. Stytz has a Bachelor of Science from the U.S. Air Force Academy, a Master of Arts from Central Missouri State University, and a Master of Science and doctorate degree from the University of Michigan.

Calculated Insight
Orlando, FL
Phone: (407) 497-4407
Fax: (703) 671-4697
E-mail: mstytz@att.net

Safety Analysis as a Software Tool

Blair T. Whatcott
Northrop Grumman Information Technology

As a software development tool, an independent software safety analysis by trained analysts reduces losses of development resources and schedule, improves product quality, and prevents costly mishaps that occur during the operational phase of the system life cycle. The key issues of an effective and efficient software safety analysis include (1) financial and managerial independence from the software development activity, (2) trained and qualified personnel to perform the analysis, and (3) a disciplined process that focuses the analysis effort, by priority, on the more safety critical areas.

Performing an independent system and software safety analysis on embedded software saves overall life-cycle cost and schedule resources, and provides a better overall product. The primary objective of safety analysis is to find and remove embedded safety related hazards in the hardware and software systems before a mishap occurs.

Finding these embedded hazards early in the development cycle reduces cost, safeguards schedules, and improves product quality [see Figure 1]. The reduction in added costs and schedule slips due to problems found late in the development cycle and the improvement in product quality justify the cost of performing an independent software safety analysis. Additionally, preventing a single catastrophic mishap by removing an embedded hazard could more than pay for the independent safety analysis effort many times over, depending upon the system.

This article identifies key terms associated with system and software safety, provides a process for performing software safety analysis, specifies the required environment for efficiently and effectively performing safety analysis, provides cost and schedule savings rationale, and identifies issues that delay or prevent effective safety analyses. Although this article emphasizes performing safety analysis on software, a thorough software safety analysis includes a system safety analysis as many of the embedded hazards occur at interfaces between system components.

When developing software systems, a tool enables a developer to build better systems quicker. The systems are more effective, more efficient, and safer. Involving an independent software safety analysis contributes to these attributes and becomes a tool that should be used in today's complex system development efforts.

Software Safety Analysis Process

An effective process for performing a software safety analysis includes four primary steps (see Figure 2):
- **Step 1.** Identify safety-critical areas and system safety hazards.
- **Step 2.** Trace implementation of safety-critical requirements to the design and its corresponding code.
- **Step 3.** Verify correct system use and implementation of safety-critical data and processing.
- **Step 4.** Track identified hazards throughout the system life cycle .

Each step is discussed in the following paragraphs.

Step 1

The first step is to list all system requirements, the relative level of safety criticality for each system requirement with respect to the other requirements, and the applicable documented hazards for safety-critical requirements. Each system/subsystem requirement must be evaluated for criticality with respect to potential hazards. The safety criticality of a requirement depends upon the identified hazards and other items such as remoteness, contributory impact, redundancies, and human intervention.

System hazards vary depending upon the function and use of the system. These hazards must be identified and documented. Sub-hazards that contribute to higher-level hazards need to be specified to a sufficient detail. An example of a hazard might be *erroneous activation of release*. Examples of corresponding contributing sub-hazards could be (1) erroneous release signal, (2) erroneous status display, and (3) malfunctioning safety lock. This list of system requirements becomes the plan that is used to perform software safety analysis.

Step 2

Using the safety criticality priority established by the list from Step 1, the second step is to evaluate requirements. This step correlates the functional requirements to (1) the top level and detailed level design descriptions and (2) the source code implementation. The code is verified for correct implementation of the requirements as well as for correct syntax and safe coding practices. This analysis also includes identifying specific safety-critical data items and processing within the reviewed code module for use in the next step. An example of a safety-critical data item could be a weapon release variable. An example of safety-critical processing could be the processing required to set or reset the release signal.

Requirements of higher safety criticality are evaluated before those of lesser criticality. The intent of safety analysis is to find the more critical hazards that would cause mishaps of higher severity. As there will always be some residual mishap risk, particularly when software is involved, the task of performing a complete and thorough software safety analysis would be both cost prohibitive as well as impossible. Consequently, items of lesser safety criticality may not be
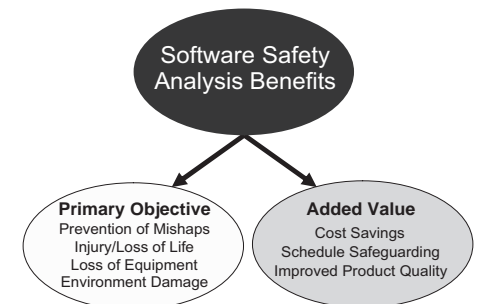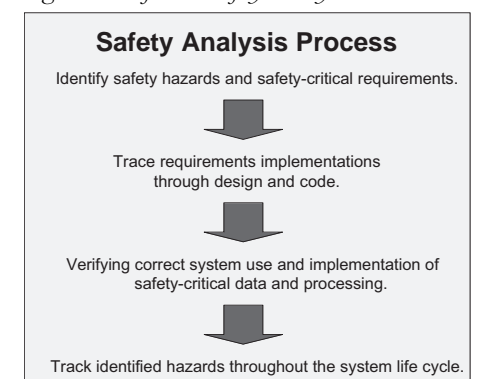
Figure 1: *Software Safety Analysis Benefits*

Figure 2: *Software Safety Analysis Process*

reviewed so that items that are more safety critical can be more deeply and completely reviewed.

Step 3

The third step is to evaluate the safety-critical data and processing identified in Step 2 in the context of the system. Particular attention is given to the interfaces between subsystems, sequencing of state changes, and timing windows of vulnerability. This type of analysis is oftentimes not given sufficient attention during software development and testing as well as peer reviews because of the added complexity and time required to be thorough.

Step 4

In the fourth step, identified hazards are documented and communicated to the development organization. The safety analysis effort tracks the identified hazard until it is removed from the software. As software hazards tend to be repeated in other areas and applications, the hazard is added to the *lessons learned* software safety analysis database. The hazards from this database, as well as hazards identified on industry *generic* safety lists, are used for training of safety analysis engineers and for performing evaluation checklists when future modifications are made to the software being analyzed or other software in other systems.

An example of a generic safety list can be found in Appendix E of the Joint Software System Safety Committee's "Software System Safety Handbook" [1]. This combined list of software-specific hazards is very large. It would be cost prohibitive to analyze every line of code against every item in the hazard list. The implementation freedom that software allows precludes an all-comprehensive automated tool that checks every line of code for every possible hazard. We have found that a trained analyst who is current with the list of software hazards is more efficient and effective in performing the safety analysis. Software utilities and tools can and are often used to help the analyst more quickly locate similar patterns, occurrences, and uses.

Software Safety Analysis Environment

To perform effective and efficient software safety analysis, an environment of three components is required: (1) financial and managerial independence from the software development activity, (2) trained and qualified personnel to perform the analysis, and (3) a disciplined process that focuses the analysis effort, by priority, on the more safety-critical areas [see Figure 3].

Each of these components is necessary for a successful software safety analysis. The absence of any undermines the effort and the strength of the other components. For example, without the financial and managerial independence from the development activity, the analyst may be directed in a way that inhibits fully performing the analysis, or the discipline process is circumvented because of management direction caused by a need to use resources in areas other than safety analysis. Similar examples can be drawn for the absence of the other components.

Financial and managerial independence ensures that specific resources will be used for performing the software safety analysis and that there is no conflict of interest between the development activity and the software safety analysis activity. An example of a conflict of interest could be the program office or development organization controlling the work of the safety analysis by direction toward or away from specific hazards or risks. The criticality list from Step 1 is the road map that identifies the priority of safety-critical areas to be reviewed. The resources are used in accordance with this priority, which means that safety-critical areas of lower priority may not be reviewed or analyzed because of the need to use resources to analyze safety-critical areas of higher priority.

An example of implementing an independent safety analysis effort would be for the program office to contract separate activities to the development organization and the software safety analysis organization. As such, reports to the program office from the software safety analysis organization are independent of the software development activity. Another example would be to have the safety office independently contract to the software safety analysis organization for work being developed by the program office that is contracted to the software development organization.

Performing a successful software safety analysis requires a technical staff that is qualified to perform safety analyses and enjoys doing this type of work. Most engineers prefer building new systems and being part of the development process of major systems. Many engineers find no interest in digging through systems to find embedded hazards that are not only difficult to find and understand, but also have not evidenced themselves. It becomes the proverbial looking for a needle in the haystack, except there is really no clue that the needle is even in the haystack or even in a number of haystacks. Finding engineers who can do this type of work and want to do it for many years is difficult. Some can do analysis for a year or so; however, the strength of a software safety analysis organization comes from analysts who have done analysis for many years on many systems.

Two pitfalls are seen when companies set up software safety organizations. First, individuals who are used to performing software safety analysis activities

Figure 3: *Safety Analysis Environment for Success*

Safety Analysis Environment for Success

Disciplined process that focuses analysis in priority on more safety-critical areas.

Financial and managerial independence from software development activity.

Trained and qualified personnel to perform analysis.

are not correctly screened with respect to ability and desire. Oftentimes, they are selected from those who have not been successful doing other code development activities because of work habit issues or lack of ability. The fallacy of doing this is that performing successful analysis on code that is generated by the development activity requires individuals that are better trained and more capable than those who have developed the code. They must be able to find embedded hazards missed by other development reviews and testing that could result in a mishap during the operational phase of the system life cycle.

The second pitfall of setting up software safety organizations is that the development activity expects that the code developer should be able to generate good code that contains no safety hazards. Consequently, the development activity either sees no value in performing an independent safety analysis or limits safety analysis resources to the point that little can be done to effectively accomplish a thorough safety analysis. They fail to understand that there is a basic difference between engineers who develop code and engineers who analyze code for embedded hazards. Engineers who develop code are success oriented. They move from the implementation of one requirement to the next. They are driven by a typically over-budget schedule and are always anxious to *catch up*. Conversely, safety engineers analyze code in the context of finding failures. They move from analyzing one module to the next only when they are convinced that there are no embedded safety concerns.

Complexity Warrants Additional Safety Analysis

With our mentality of getting the most out of software development budgets combined with the mindset that developers can generate hazard-free code, managers pressure software developers to generate code faster and more efficiently. They insist that better processes, pride of workmanship, better compilers and development environment tools, code walk-throughs, and peer reviews should sufficiently guarantee safe code. It is true that compilers and development environment tools are becoming more powerful, but at the same time they are becoming more complex. This additional complexity warrants additional safety analysis. It is true that code walk-throughs, peer reviews, and other process improvements result in better code; however, they rely upon peers who are also behind

---

Safety Terminology

For the purposes of this article, the following safety-related terms are provided from [2].

- A **mishap** is an unplanned event that results in death, injury, occupational illness, equipment or property damage or the loss of, or environmental damage.
- **Hazards** are conditions that cause a mishap.
- The **ultimate goal of a system safety program** is to design systems that contain no hazards. However, since the nature of most complex systems makes it impossible or impractical to design them completely hazard-free, a successful system safety program often provides a system design where there exist no hazards resulting in an unacceptable level of mishap risk.
- **Mishap risk** is an expression of the possibility/impact of a mishap in terms of hazard severity and hazard probability.
- **Residual mishap risk** is the remaining mishap risk after all mitigation techniques (techniques used to remove or lessen the hazard) have been implemented or exhausted.
- **Safety** is the freedom from hazards, which cause death, injury, occupational illness, equipment or property damage or loss, or environmental damage.
- The **objective of a safety analysis** is to achieve acceptable mishap risk through a documented systematic approach to hazard analysis, risks assessment, and risk management.
- **System safety** is the application of engineering and management principles, criteria, and techniques to achieve acceptable mishap risk, within the constraints of operational effectiveness and suitability, time, and cost, throughout all phases of the system life cycle.

---

schedule, over-budget, and anxious to get their own work done. These distractions lessen the effectiveness of the peer review. Additionally, peer reviews tend to focus on the module level and place less emphasis at the system level where many of the embedded safety hazards reside.

Software developers must be safety conscious as they develop code. However, in light of the above and their success orientation, developers continue to introduce embedded hazards in the software development process; it is very difficult for them to see their own errors. E-mails are a vivid example. E-mail authors re-read their own e-mails over and over to verify correctness. They send them out only to later find a glaring error in the most awkward place that they missed during multiple reviews. Some well-known examples of software failures resulting in mishaps are described in Appendix F of the Joint Software System Safety Committee's "Software System Safety Handbook" [1].

Engineers who effectively analyze code for embedded hazards are convinced that all software contains embedded hazards and that it is only a matter of time and circumstance before the hazard(s) causes a mishap. The quality and quantity of analysis is a function of the analyst's safety experience and understanding of the code under inspection within the context of the system. Tangible products of the analysis may be

misleading as amount and quality of product does not necessarily prove that the right analysis was performed. On the other hand, the tangible products of a development effort do prove the efforts of the development engineer.

From the development activity perspective, if the code successfully performs its intended function and matches documented code standards, then return on investment is evident. Failure to identify embedded hazards does not confirm that quality analysis has not been performed any more than the identification of some embedded hazards ensures that all hazards have been found. The analyst decides the correct amount of effort spent in the analysis of a safety-critical area for hazards without evidence of their existence based on his or her experience and understanding.

In summary, development engineers are good at building new systems in the context of a driving schedule. Software safety engineers are good at evaluating code for embedded hazards. Requiring development engineers to constantly evaluate their code with the understanding that *something is wrong and there are embedded hazards* seriously takes away from the success orientation that enables forward progress. Software safety analysis engineers are attuned to the identification of embedded hazards and the amount of resources required to fully analyze a safety-critical area of code. Just

as letting off an automobile's gas pedal does perform some slowing, and letting off the brake pedal allows continued movement, both the gas pedal and the brake pedal are required for efficient handling. A combination of development engineers and software safety engineers in an independent environment provides a product that is synergistically more than if either were to do both tasks.

The software safety analysis process combines the people and the resources to produce the most effective and efficient product possible. The process ensures that priorities are followed, products are produced, and schedules are met. The four primary steps of a software safety analysis process have been described. Necessary products of the safety analysis include a criticality analysis report from Step 1, problem reports from all steps, and a software safety analysis report – including testing and analysis summaries – from Step 2 through Step 4 of the process. When a thorough and conscientious software safety analysis is complete, and safety hazards have been identified and removed, the resulting summary report becomes a tangible product that indicates with a high level of confidence that the examined software will not be the source of a system mishap.

Issues That Hinder Software Safety Efforts

There are many reasons why organizations mistakenly choose not to include a software safety analysis activity early in the code development cycle (see Table 1). These include the following:

1. Organizations erroneously believe that performing software safety analysis only needs to be done when code has been generated. They believe that they can conserve resources during the requirements definition and design disclosure phases by waiting until code is released to involve the software safety analysis effort. They fail to understand the importance of evaluating system and functional requirements with respect to safety prior to design, and of evaluating the design disclosure with respect to safety prior to coding. Safety concerns found during the implementation phase after the code has been generated require re-evaluation of the requirements, redesign, and recoding. This results in wasted resources and schedule slips because of the necessary review and rework. This is further impacted by the software safety analyst's need for time to become familiar with the function, requirements, design, and code of the software under analysis. If this need is put off until code is released, then safety concerns are, consequently, identified later in the implementation phase, resulting in additional wasted resources because testing must also be repeated due to reworked requirements, design, and code.

2. Government organizations find it difficult and time consuming to establish a contract with an independent organization to do software safety analysis. It is important to start the process early to take into account the lead times as well as the need for either contracting directly with the software safety analysis company or using a contract vehicle already in place by the contractor.

3. The organization erroneously believes that a good code development process will preclude all embedded safety hazards. Mishaps caused by software occur in fielded systems that were developed under good processes. As described earlier, an independent software safety analysis can find embedded hazards and prevent mishaps when trained and experienced analysts are used and the software safety analysis process is followed.

4. Organizations fail to factor into their budget the software safety analysis activity when cost projections are supplied to planning activities. Upon program execution, they severely limit or do not fund software safety activities because of the difficulty of finding unbudgeted resources to cover safety. Including software safety analysis activities in the master budget plan is critical to software safety.

5. The lack of mishap evidence gives the program manager a false impression of the safety state of the software being developed. If an embedded hazard is found and removed, there is no evidence that the mishap would have ever occurred. Embedded hazards cause catastrophic mishaps only when a set of combining circumstances simultaneously occurs. A thorough analysis covers areas and combinations of events that are either difficult to test or are not tested because of limitations due to test time and tester expertise.

6. Organizations have difficulty finding

Table 1: *Mistaken Reasons Why Software Safety Is Not Included During Early Phases of the System Development Life Cycle*

Mistaken Reasons Why Software Safety Is Not Included During Early Phases of System Software Development Life Cycle	
Reason	**Actual Need**
Conserve funds because there is no code to review.	Early evaluation of requirements and design precludes costly coding and testing errors.
Difficulty establishing a contract with an independent organization to do safety.	Most major defense contractors have General Services Administration-type contracts that could support safety efforts.
Misconception that good coding processes preclude embedded safety hazards.	Success orientation of development engineers results in missed errors; development environment pressures prevent thorough system and interface analysis.
Failure to budget in software safety analysis activities.	Software safety analysis needs to be budgeted independently of development activity budgets.
Lack of mishap evidence if hazard found and removed.	The objective of software safety is to remove hazard before mishap; prevention of one catastrophic mishap more than pays for safety analysis effort.
Lack of software safety analysis expertise and processes.	Evaluate potential safety analysis organizations on track record, processes, and staff.
Problems found in safety analysis cause additional work impacts.	Early identification of safety problems saves resources by preventing redesign, recode, retest, and prevents mishap.

software safety analysis expertise and processes. As described earlier, effective analysis is a function of the expertise and experience of the analyst. Qualified sources for software safety expertise will probably be more costly because of the need to employ this level of expertise and experience.

7. Organizations are concerned that problems found by software safety analysis will cause additional work that impacts schedule and resource needs. Reputable organizations do not generate unsafe software. However, because of the nature of embedded hazards that result in mishaps, there is always the concern that large amounts of resources are spent to prevent mishaps that have a very low probability of occurring. These organizations fail to understand that providing a small level of software safety analysis can greatly lower the probability of a mishap occurring.

Each of these mistaken reasons is real. Together they may discourage using software safety analysis as a tool to generate a better product for less cost. Finding these embedded hazards early in the development cycle reduces cost, safeguards schedules, and improves product quality. Our experience shows that a requirements problem that is not found until the test phase of the software development cycle results in the loss of 70 percent of the time used to design, code, and test the implementation of that requirement.

Summary

We live in a world that is averse to unsafe conditions. We also live in a world that applies heavy pressure to building the *better and faster* more efficiently. The conflicts between these two mindsets are profit and risk. The courts of the land insist daily upon the responsibility of the product provider. Flashy packaging and brand-name recognition oftentimes erroneously instill within us a false sense of trust. And if we are harmed, our loss of productivity and capability demands compensation in order to survive.

Software safety analysis as a tool results in a safer and better product at a cost and schedule savings. Early involvement is critical to an efficient and effective analysis effort. Software requirements hazards will be found and removed during the requirements phase. Hazards found during the other software development phases will be found during the correct phase, preventing loss of
resources and schedule.

Software development teams want to generate a quality product, but are hesitant to have independent activities perform analysis on their product. A change of mindset will result in a synergistic team that produces a superior product. Development engineers will be able to do what they do best in a success-oriented environment within their resources and schedules. Software safety analysis engineers will provide the necessary checks and balances that result in a superior product, free of embedded hazards. When these work as a team, software development will cost less and be provided on schedule in our world of continuous change and improvement.◆

References

1. Joint Software System Safety Committee. Software System Safety Handbook. Washington, D.C.: Department of Defense, Dec. 1999 <www.egginc.com/dahlgren/files/ssshandbook.pdf>.
2. Department of Defense. "Standard Practice for System Safety." MIL-STD 882D. Washington, D.C.: DoD, 10 Feb. 2000 <www.safetycenter.navy.mil/instructions/osh/milstd882d.pdf>.

About the Author

Blair T. Whatcott is the lead engineer and program manager of the Software and System Safety Analysis Program Office in the Information Solutions Department of Northrop Grumman Information Technology. He has managed and been lead engineer on independent verification and validation and software safety analysis projects for more than 18 years. These projects have focused on detailed analysis and testing of embedded software in military aircraft and weapon systems. He has a bachelor's degree in electrical engineering from Brigham Young University, Provo, Utah.

Northrop Grumman
Information Technology
1530 N Layton Hills PKWY
STE 200
Layton, UT 84041-5683
Phone: (801) 773-5274 ext. 13
Fax: (801) 773-5262
E-mail: blair.whatcott@ngc.com

Three Essential Tools for Stable Development©

Andy Hunt and Dave Thomas
The Pragmatic Programmers, LLC

Three basic practices make the difference between a software project that succeeds and one that fails. These practices support and reinforce each other; when done properly, they form an interlocking safety net to help ensure success and prevent common project disasters. However, few development teams in the United States use these proven techniques, and even fewer use them correctly.

Many software projects that fail seem to fail for very similar reasons. After observing – and helping – many of these ailing projects over the past couple of decades, it seems clear to us that a majority of common problems can be traced back to a lack of three very basic practices. Fortunately, these three practices are easy and relatively inexpensive to adopt. It does not require a large-scale, expensive, or bureaucratic effort; with just these practices in place, your team can work at top speed with increased parallelism. You will never lose precious work, and you will know immediately when the development starts to veer off-track in time to correct it, cheaply and easily.

The three basic practices that we have identified as being the most crucial are *version control*, *unit testing*, and *automation*. Version control is an obvious *best practice*, yet nearly 40 percent of software projects in the United States do not use any form of version control for their source code files [1]. The motto of these shops seems to be *last one in wins*. That is, they will use a shared drive of some sort and hope that no one overwrites their changes as the software evolves. Hope is a pretty poor methodology, and these teams regularly lose precious work. Developers begin to fear making any changes at all, in case they accidentally make the system worse. Of course, this fear becomes a self-fulfilling prophecy as necessary changes are neglected and the system begins to degrade.

Unit testing is a coding technique for programmers so they can verify that the code they just wrote actually does something akin to their intent. It may or may not fulfill the requirements, but that is a separate question: If the code does not do what the programmer thought it did, then any further testing or validation is both

© 2004 The Pragmatic Programmers, LLC. Portions of this article adapted from "Pragmatic Unit Testing in Java With JUnit," by Andy Hunt and Dave Thomas (Volume II of the Pragmatic Starter Kit), published by the Pragmatic Bookshelf and Copyright © 2003, 2004 The Pragmatic Programmers, LLC. Reprinted with permission.

meaningless and a large waste of time and money (two items that are in short supply to begin with). Developer-centric unit testing is a great way to introduce basic regression testing, create more modularized code that is easier to maintain, and ensure that new work does not break existing work. Despite the effectiveness of this technique in both improving design and identifying and preventing defects (aka bugs), 76 percent of companies in the United States do not even try it [2].

Automation is a catchall category that includes regular, unattended project builds, including regression tests and push-button convenience for day-to-day activities. Regular builds ensure that the product can be built to catch simple mistakes early and easily, when fixing them is the cheapest. When implemented properly, it is as if you have an ever-vigilant guardian looking over your shoulder, warning you as soon as there is a problem. Incredibly, some 70 percent of projects in the United States do not have any sort of daily build [2]. By the time they discover a problem, it has metastasized into a much larger and potentially fatal problem.

We will briefly examine each of these areas, with an in-depth look at unit testing in particular. We will outline the important ideas, synergies, and caveats for each of these practices so your team can either begin using them or improve your current use of them.

Version Control

Everyone can agree that version control is a best practice but even with it in place, is it being used effectively? Ask yourself these questions: Can you re-create your software exactly as it existed on January 8? When a bug is found that affects multiple versions of your released software, can your team fix it just once, and then apply that fix to the different versions automatically? Can a developer quickly back out of a bad piece of code?

There is more to version control than just keeping track of files. But before we

proceed, we need to define some simple terminology: We use *check-in* to mean that a programmer has submitted his or her changes to the version control system. We use *checkout* to refer to getting a personal version of code from the version control system into a local working area.

When a programmer checks in code, it is now potentially available to the rest of the team. As such, it is only polite to ensure that this new code actually compiles successfully; it should be accompanied by unit tests (more on this later), and those tests should pass. All the other passing tests in the system should continue to pass as well – if they suddenly fail, then you can easily trace the failure to the new code that was introduced.

It is far easier to track down these sort of problems right at the point of creation instead of days, weeks, or even months later. To exploit this effect, you must allow and encourage frequent check-ins of code multiple times per day. It is not unusual to see team members check-in code 10-20 times a day. It *is* unusual – and very dangerous – to allow a programmer to go a few days or a week or more without checking in code.

Because check-ins occur so frequently, these and other day-to-day operations must be very fast and low ceremony. A check-in or checkout of code should not take more than five to 15 seconds in general. If it takes an hour, people will not do it, and you have lost the advantage.

Now some people get a little nervous when they read this part. They fret that all of this code is being dumped into the system without being reviewed, tested by QA, audited, or whatever else their methodology or environment demands. They are rightfully concerned that this code is not yet ready to be part of a release. Nonetheless, it must still be in the version control system so that it is protected.

Most version control systems provide a mechanism to differentiate ongoing development changes from official release candidates. Some feature explicit *promotion*

22 CROSSTALK The Journal of Defense Software Engineering

November 2004

commands to allow this. You can accomplish the same thing in other systems by using tags (or version labels) to identify stable release versions of source code as opposed to code that is in progress.

Regardless of the mechanism, it must be an easy operation to promote development changes to an official release status. On the other side of the coin, you need to be able to back out changes and any disastrous new code when needed.

Finally, you need to be able to re-create any product built at any previous point in time. This ability to go back in time is crucial for effective debugging and problem solving (just think of any developer who starts a discussion with, "Well, it used to work").

Commercial and freely available version control systems vary in complexity, features, and ease of administration. But one feature in particular is worth examining: whether it supports strict locking or optimistic locking. In systems under strict locking, only one person can edit a file at a time. While that sounds like a good idea, it turns out to be unduly restrictive in practice. We favor the Concurrent Version System <www.cvshome.org> described in [3].

You may find you can increase parallelism and efficiency in your team by using a system that features optimistic locking. In these systems, multiple people can edit the same source code file simultaneously. The system uses conflict-resolution algorithms to merge the disparate changes together in a sensible manner. Ninety-nine percent of the time it works perfectly without intervention. Occasionally, however, there is a conflict that must be addressed manually. At no point is anyone's work in danger of being lost, and it ends up being much more efficient to coordinate just these few conflicts by hand instead of having everyone coordinate *every* change with the rest of the team.

Unit Testing

When a developer makes a change to the code on your project, what feedback is available? Does the developer have any way of knowing if the new code broke anything else? Better still, how do *you* know if any developer has broken anything today? A system of automated unit tests will give you this information in real-time.

Programming languages are notorious for doing exactly what programmers say, not what they mean. Like a petulant child that takes your expressions completely literally, the computer follows our instructions to the letter, with no regard at all to our intent. Technology has yet to produce

the compiler that implements with *do what I mean, not what I say.*

So in keeping with the idea of finding and fixing problems as soon as they occur, you want programmers to use unit tests (or checked examples) to verify the computer's literal interpretation of their commands. It is really no different from following through with a subordinate to verify that a delegated task was performed – except that instead of just checking once, automated unit tests will check and recheck every time any code is changed.

There are some requirements to using this style of development, however:

- The code base must be decoupled enough to allow testing. When code is tightly coupled, it is very difficult to test individual pieces in isolation, and harder to devise unit tests that exercise specific areas of functionality. Well-written code, on the other hand, is easy to test. If your team finds that the code is difficult to test, then take that as a warning sign that the code is in serious trouble to begin with.
- Only check-in tested code. As we mentioned above, checking-in foists a programmer's code onto the rest of the team. Once it is available to everyone, then the whole team will begin to rely on it. Because of this reliance, all code that is checked in must pass its own tests.
- In addition to passing its own tests, the programmer checking in the code must ensure nothing else breaks, either. This simple regression helps prevent that frustrating feeling of *one step forward, two steps back* that becomes commonplace when code fixes cause collateral damage to other parts of the code base. Usually these bugs then require fixes, which in turn cause more damage, and so on. The discipline of keeping all the tests running all the time prevents that particular death-spiral.
- There should be at least as much test code as production code. You might think that is excessive, but it is really just a question of where the value of the system resides. We firmly believe the code that implements the system is not where the value of your intellectual property lies. Code can be rewritten and replaced, and the new code (even an entirely new system) can be verified against the existing tests. Now the most precise specification of the system is in executable form – the unit tests. The learning and experience that goes into creating the unit tests is invaluable, and the tests themselves are the best expression we have of that

knowledge.

We will look at implementing unit tests (aka checked examples) in much greater detail later in this article.

Automation

An old saying goes the *cobbler's children have no shoes.* This saying is particularly appropriate for our use of software tools during software development. We see teams routinely waste time using manual procedures that could easily be automated.

Everyone clamors for software development to be more defined and repeatable. Well, the *design* and implementation of software probably cannot be made repeatable any more than you could make the process of making hit movies repeatable. But the *production* of software is another matter entirely.

The process of taking source code files, bits of eXtensible Markup Language, libraries, and other resources and producing an executable for the end user should be precisely repeatable. Given the same inputs, you want the same outputs, every time, without excuses. In combination with version control, you want to be able to go back in time and reproduce that same pile of bits that you would have produced on January 8 just as easily. That comes in very handy should the Department of Justice ask for it politely, or a frustrated customer asks for it somewhat less politely to work around some outstanding bug.

The rule we try to adopt is that any manual process that is repeated twice is likely to be repeated a third time – or more – so it needs to be encapsulated within a shell script, batch file, piece of Java code, Job Control Language, or whatever.

Unit tests, as well as functional and acceptance tests, should be run automatically as well as be part of the build process. You will probably want to run the unit tests (which should execute very quickly) with every build; automatic functional and acceptance tests might take longer and you may only want to run those once a week, or when convenient.

You see, not only does automation make developer's lives easier by providing push-button convenience, it helps keep the feedback coming by constantly checking the state of the software. Automated builds are constantly asking two questions: Does the software build correctly? Do all the tests still pass a basic regression? With the computer performing these checks regularly, developers do not have to. Problems can be identified as soon as they happen, and the appropriate developer or team lead can be notified immediately of

the problem [4]. Problems can be fixed quickly, before they have a chance to cause any additional damage. That is the benefit we want from automation.

Finally, consider how the build communicates to the development team and its management. Does the team lead look at the latest results in some log file and then report status to management? Does not that constitute a manual process? It is relatively easy to set up visual display devices, ranging from liquid crystal display screens to bubbling lava-style lamps to the new and popular Ambient Orb [4].

Synergy

These three practices interlock to provide a genuine safety net for developers. Version control is the foundation. Unit tests and scripts for automation are under version control, but version control needs automation to be effective. Unit testing needs both version control and automation.

With the combination, developers can better afford to take chances, experiment, and find the best solutions. The Rule of Three says that if you have not proposed at least three solutions to a problem then you have not thought about it hard enough. With this set of practices in place, developers can realistically try out a number of different solutions to a problem: Version control will keep them separate, and unit testing will help confirm the viability of each solution. All this with plenty of automated support, including continuous, ongoing checks ensures that the team does not wander too far off into the woods. This is how modern, successful software development is done.

Unit Testing With Your Right-BICEP

You can strengthen your organization's testing skills by looking at six specific areas of code that may need unit tests. These areas are remembered easily using the mnemonic Right-BICEP [5]:

Right   Are the results *right*?
B   Are all the *boundary* conditions correct?
I   Can you check *inverse* relationships?
C   Can you *cross-check* results using other means?
E   Can you force *error* conditions to happen?
P   Are *performance* characteristics within bounds?

Are the Results Right?

The first and most obvious area to test is

simply to see if the expected results are right – to validate the results. These are usually the *easy* tests, as they represent the answer to the key question: If the code ran correctly, how would I know? Here is an example of how being forced to think about testing helps developers code better: If this question cannot be answered satisfactorily, then writing the code – or the test – may be a complete waste of time.

"But wait," you cry out, "that does not sound very agile! What if the requirements are vague or incomplete? Does that mean we can't write code until all the requirements are firm?" No, it does not at all. If the requirements are truly not yet known, or not yet complete, you can always make some assumptions as a stake in the ground. They may not be correct from the user's point of view (or anyone else on the planet), but they let the team continue to develop. And, because you have written a test based on your assumption, you have now documented it – nothing is implicit.

Of course, you must then arrange for feedback with users or sponsors to fine-tune your assumptions. The definition of *correct* may change over the lifetime of the code in question, but at any point, you should be able to prove that it is doing what you think it ought.

Boundary Conditions

Identifying boundary conditions is one of the most valuable parts of unit testing because this is where most bugs generally live – at the edges. Some conditions you might want to think about include the following:

* Totally bogus or inconsistent input values such as a file name of !*W:X\\{\&Gi/w$>$g/h\#WQ@.
* Badly formatted data such as an e-mail address without a top-level domain <fred@foobar>.
* Empty or missing values such as 0, 0.0, "", or null.
* Values far in excess of reasonable expectations such as a person's age of 10,000 years.
* Duplicates in lists that should not have duplicates.
* Ordered lists that are not in order and vice-versa. Try handing a pre-sorted list to a sort algorithm, for instance, or even a reverse-sorted list.
* Things that arrive out of order, or happen out of expected order such as trying to print a document before logging in, for instance.

An easy way to think of possible boundary conditions is to remember the acronym CORRECT. For each of these items, consider whether or not similar con-

ditions may exist in your method that you want to test, and what might happen if these conditions were violated [4]:

* **Conformance.** Does the value conform to an expected format?
* **Ordering.** Is the set of values ordered or unordered as appropriate?
* **Range.** Is the value within reasonable minimum and maximum values?
* **Reference.** Does the code reference anything external that is not under direct control of the code itself?
* **Existence.** Does the value exist (e.g., is non-null, non-zero, present in a set, etc.)?
* **Cardinality.** Are there exactly enough values?
* **Time (absolute and relative).** Is everything happening in order? At the right time? In time?

Check Inverse Relationships

Some methods can be checked by applying their logical inverse. For instance developers might check a method that calculates a square root by squaring the result, and testing that it is tolerably close to the original number. They might also check that some data was successfully inserted into a database by then searching for it, and so on.

Be cautious when the same person has written both the original routine and its inverse, as some bugs might be masked by a common error in both routines. Where possible, use a different source for the inverse test. In the square root example, we might use regular multiplication to test our method. For the database search, we will probably use a vendor-provided search routine to test our insertion.

Cross-Check Using Other Means

Developers might also be able to cross-check results of their method using different means. Usually there is more than one way to calculate some quantity; we might pick one algorithm over the others because it performs better or has other desirable characteristics. That is the one we will use in production, but we can use one of the other versions to cross-check our results in the test system. This technique is especially helpful when there is a proven, known way of accomplishing the task that happens to be too slow or too inflexible to use in production code.

Another way of looking at this is to use different pieces of data from the code itself to make sure they all *add up*. For instance, suppose you had some sort of system that automated a lending library. In this system, the number of copies of a particular book should always balance. That is, the number of copies that are

checked out plus the number of copies sitting on the shelves should always equal the total number of copies in the collection. These are separate pieces of data, and may even be managed by different pieces of code, but they still have to agree and so can be used to cross-check one another.

Force Error Conditions

In the real world, errors happen. Disks fill up, network lines drop, e-mail goes into a black hole, and programs crash. You should be able to test that code handles all of these real-world problems by forcing errors to occur.

That is easy enough to do with invalid parameters and the like, but to simulate specific network errors – without unplugging any cables – takes some special techniques, including using mock objects.

In movie and television production, crews will often use *stand-ins*, or doubles, for the real actors. In particular, while the crews are setting up the lights and camera angles, they will use *lighting doubles*: inexpensive, unimportant people who are about the same height and complexion as the very expensive, important actors who remain safely lounging in their luxurious trailers.

The crew then tests their setup with the lighting doubles, measuring the distance from the camera to the stand-in's nose, adjusting the lighting until there are no unwanted shadows, and so on, while the obedient stand-in just stands there and does not whine or complain about *lacking motivation* for their character in this scene.

What you can do in unit testing is similar to the use of lighting doubles in the movies: Use a cheap stand-in that is kind of close to the real thing, at least superficially, but that will be easier to work with for your purposes.

Performance Characteristics

One area that might prove beneficial to examine is performance characteristics – not performance itself, but trends as input sizes grow, as problems become more complex, and so on. Why? We have all experienced applications that work fine for a year or so, but suddenly and inexplicably slow to a crawl. Often, this is the result of a silly error or oversight: A database administrator changed the indexing structure in the database, or a developer typed an extra zero into a loop counter.

What we would like to achieve is a quick regression test of performance characteristics. We want to do this regularly, every day at least, so that if we have inadvertently introduced a performance problem we will know about it sooner

rather than later (because the nearer in time you are to the change that introduced the problem, the easier it is to work through the list of things that may have caused that problem).

So, to avoid shipping software with unsuspected performance problems, teams should consider writing some rough tests just to make sure that the performance curve remains stable. For instance, suppose the team is working on a component that lets users browse the Web from within their application. Part of the requirement is to filter out access to Web sites that we wish to block. The code works fine with a few dozen sample sites, but will it work as well with 10,000? 100,000? We can write a unit test to find out.

This gives us some assurance that we are still meeting performance targets. But because this one test takes six to seven seconds to run, we may not want to run it every time. As long as we run it (say) nightly, we will quickly be alerted to any problems we may introduce while there is still time to fix them.

Getting Started

All of the software tools mentioned in this article are freely available on the Web. To get started using these practices effectively, we recommend following this sequence:
1. Get everything into version control.
2. Arrange for automatic, daily builds. Increase these to multiple times per day or continuously as soon as the process begins to work smoothly.
3. Start writing unit tests for new code.

Where needed, add some unit tests to existing code (but be pragmatic about it; only add tests if they will really help, not just for the sake of completeness).
4. Add the unit tests to the scheduled builds.

You can begin right away. Fire up that Web browser and start downloading some software if you do not already have it. These ideas will not fix all the problems on your project, of course, but they will provide your project with a firm footing so you can concentrate on the truly difficult problems.◆

References

1. Zeichick, Alan. "Debuggers, Source Control Keys to Quality." <u>Software Development Times</u> 1 Mar. 2002.
2. Cusumano, Michael, et al. "A Global Survey of Software Development Practices." Paper 178. MIT Sloan School of Management, June 2003.
3. Thomas, Dave, and Andy Hunt. <u>Pragmatic Version Control With CVS</u>. Raleigh, NC: Pragmatic Bookshelf, 2003 <www.PragmaticBookshelf.com>.
4. Clark, Mike. <u>Pragmatic Project Automation</u>. Raleigh, NC: Pragmatic Bookshelf, 2004 <www.PragmaticBookshelf.com>.
5. Hunt, Andy, and Dave Thomas. <u>Pragmatic Unit Testing in Java With JUnit</u>. Raleigh, NC: Pragmatic Bookshelf, 2003. (Also available in a C# version) <www.PragmaticBookshelf.com>.

About the Authors

Andy Hunt is an avid woodworker and musician, but curiously, he is more in demand as a consultant. He has worked in telecommunications, banking, financial services, and utilities, as well as more exotic fields such as medical imaging and graphic arts. Hunt is author of many articles, columns and books, and co-author of "The Pragmatic Programmer."

Dave Thomas likes to fly single-engine airplanes and pays for his habit by finding elegant solutions to difficult problems, consulting in areas as diverse as aerospace, banking, financial services, telecommunications, travel and transport, and the Internet. Thomas is author of many articles, columns and books, and co-author of "The Pragmatic Programmer."

The Pragmatic Programmers, LLC
9650 Strickland RD
STE 103-255
Raleigh, NC 27615
Phone: (800) 699-7764
E-mail: andy@pragmatic
programmer.com

The Pragmatic Programmers, LLC
P.O. Box 293325
Lewisville, TX 75029
Phone: (972) 539-7832
E-mail: dave@pragmatic
programmer.com

CROSSTALK
The Journal of Defense Software Engineering

Get Your Free Subscription

Fill out and send us this form.

OO-ALC/MASE
6022 FIR AVE
BLDG 1238
HILL AFB, UT 84056-5820

FAX: (801) 777-8069 DSN: 777-8069

PHONE: (801) 775-5555 DSN: 775-5555

Or request online at www.stsc.hill.af.mil

NAME:_____

RANK/GRADE:_____

POSITION/TITLE:_____

ORGANIZATION:_____

ADDRESS:_____

_____

BASE/CITY:_____

STATE:_____ZIP:_____

PHONE:(____)_____

FAX:(____)_____

E-MAIL:_____

CHECK BOX(ES) TO REQUEST BACK ISSUES:

AUG2003 ☐ NETWORK-CENTRIC ARCHT.
SEPT2003 ☐ DEFECT MANAGEMENT
OCT2003 ☐ INFORMATION SHARING
NOV2003 ☐ DEV. OF REAL-TIME SW
DEC2003 ☐ MANAGEMENT BASICS
MAR2004 ☐ SW PROCESS IMPROVEMENT
APR2004 ☐ ACQUISITION
MAY2004 ☐ TECH.: PROTECTING AMER.
JUN2004 ☐ ASSESSMENT AND CERT.
JULY2004 ☐ TOP 5 PROJECTS
AUG2004 ☐ SYSTEMS APPROACH
SEPT2004 ☐ SOFTWARE EDGE
OCT2004 ☐ PROJECT MANAGEMENT

TO REQUEST BACK ISSUES ON TOPICS NOT LISTED ABOVE, PLEASE CONTACT KAREN RASMUSSEN AT <STSC.CUSTOMERSERVICE@ HILL.AF.MIL>.

WEB SITES

The Agile Alliance
www.agilealliance.com/home
The Agile Alliance is a non-profit organization dedicated to promoting the concepts of agile software development, and helping organizations adopt those concepts. The site features an extensive library of articles about agile processes and agile development.

COTS Journal
www.cotsjournalonline.com
COTS Journal is a technology-in-context magazine that looks at any embedded technology anywhere it exists. Its editors assess the applicability of the world's best embedded research and standards, methodologies, and products for government, military, and aerospace applications. *COTS Journal* provides the industry with technical material to help readers design and build embedded computers for the military – whether for benign applications or for the most rugged, mission-critical jobs.

Concurrent Versions System
www.cvshome.org
This site is dedicated to supporting the community around the Concurrent Versions System (CVS). The CVS is the dominant open-source network-transparent version control system. CVS is useful for everyone from individual developers to large, distributed teams for the following:

• Its client-server access method lets developers access the latest code from anywhere there is an Internet connection.

• Its unreserved checkout model to version control avoids artificial conflicts common with the exclusive checkout model.

Control Chaos.com
http://controlchaos.com
Control Chaos.com is home to Scrum, an agile, lightweight process that can be used to manage and control software and product development using iterative, incremental practices. Wrapping existing engineering practices, including eXtreme Programming and RUP, Scrum generates the benefits of agile development with the advantages of a simple implementation. Scrum significantly increases productivity and reduces time to benefits while facilitating adaptive, empirical systems development. Advanced Development Methods, Inc. maintains this Web site to provide information, news, references, and a cookbook description of Scrum.

Air Force Research Laboratory/Information Directorate
www.rl.af.mil
The Air Force Research Laboratory/Information Directorate (AFRL/IF) is a confluence of information specialists, electrical and computer engineers, computer scientists, mathematicians, physicists, and a supporting staff. The AFRL/IF develops systems, concepts, and technologies to enhance the Air Force's capability to successfully meet the challenges of the information age. It develops and integrates programs to acquire data and to find better ways to store, process, and fuse data to make it into information. The AFRL/IF creates the means to deliver and present tailored information to allow the military decision maker to have the total sphere of information needed for successful operations worldwide.

MIT System Safety and Software Safety Research
http://sunnyday.mit.edu/safety.html
The goal of the Massachusetts Institute of Technology's (MIT) System Safety and Software Safety Research project is to develop a theoretical foundation for safety and a methodology for building safety-critical systems built upon that foundation. The methodology includes special management structures and procedures, system hazard analysis, software hazard analysis, requirements modeling and analysis for completeness and safety, design for safety, design of human-machine interaction, verification (both testing and code analysis), operational feedback, and change analysis.

Your Quality Data Is Talking – Are You Listening?

David B. Putman
Ogden-Air Logistics Center

The transition from defect detection and removal activities to defect prevention activities may not be as smooth as you would like. You may start asking, "Where do I start?" Or, you may have the feeling that you are not getting much benefit from your defect prevention activities. You may also find yourself faced with a need to explore, evaluate, and adopt new metrics. This article discusses some quantitative (non-statistical process control [SPC]) methods for looking at your data; I will show the results of applying SPC to the same information, and provide a few "what next" options. The intent of this article is to provide process improvement team members, program managers, and supervisors with ideas for defect prevention metrics to help them identify and analyze problem areas and to help them prioritize and plan their defect prevention activities. I have chosen to avoid discussing complex mathematical algorithms in favor of providing charts to aid the reader in participating in brainstorming activities to identify metrics they will find useful for their situation.

You know it is impossible to fix every problem at once so you review the defect information looking for something that will jump out and say, "Fix me." During your review of the data, you find an item that grabs your attention. You are confident that you can reduce type *xyz* defects by 90 percent simply by providing the organization with an annual eight-hour refresher-training course. You estimate that it will cost $20,000 to develop a formal training course, and you get management approval to implement the idea. A few weeks later, you provide the first eight-hour training course to a team of 50 employees.

Six months later you analyze the data and, to your credit, you exceeded your goal: Type xyz defects were reduced by 95 percent. Unfortunately, you learn that your savings in development and rework costs is significantly less than the annual costs for the training. You also realize that all type xyz defects were detected internally and none were ever released to the customer. In order to maintain your integrity, you brief management of your findings and recommend discontinuing the annual eight-hour training course.

You cannot try to solve every type of defect at once so clearly you need a way of prioritizing your efforts. You also need a way of evaluating the possible solutions (cost versus benefit) to determine the most effective solution. This article is aimed at giving the reader some ideas on what type of defect information should be captured, and ways to present that data. Armed with the proper information, a defect prevention team will be able to prioritize its efforts, evaluate the effectiveness of the proposed solutions, and determine the proper corrective action.

Quantitative (Non-Statistical Process Control) Data Analysis

As our Software Engineering Division at the Ogden-Air Logistics Center increased its focus on defect prevention activities, the Extended Software Engineering Process Group (ESEPG) found that it was not receiving much utility from its existing quality metrics. At the request of the ESEPG, I began analyzing its data in an effort to recommend some potential metrics that would facilitate defect prevention activities.

In my data analysis, I explored a variety of ways to show the data in order to provide the ESEPG with the ability to prioritize its efforts. Our group had collected a vast amount of information, so the first task was to develop appropriate filters to give me a better ability to extract the data in a manner that would facilitate the analysis. My first look at the information was by the category and severity of the defect as shown in Figure 1. If defects of a high severity were getting through the process, then this would be a logical starting point for defect prevention activities.

As seen in Figure 1, almost all of the recent defects were identified as being a minor severity. At this point, I changed the filters to extract the information for 18 different categories and types of defects, and then again for 19 different categories and locations for the defects. Table 1 (page 28) provides an example of how each defect is characterized by category, type, and location.

The documentation defects analysis showed that typographical errors in the engineering documentation used to maintain the product were the most common defect type found during peer reviews. I then began to perform a sim-

Figure 1: *Quantity of Defects By Severity and Category*

All Defects Found in the Last 12 Months By Severity and Category
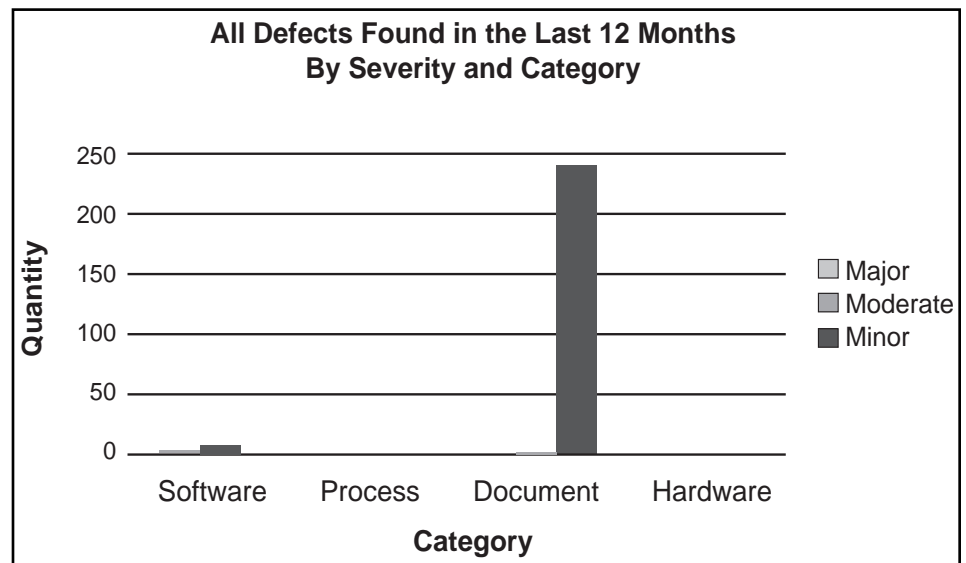
Category	Type	Location
Software	Syntax	Source Code
Software	Typographical	Source Code (e.g., comment)
Documentation	Typographical	User's Manual
Documentation	Typographical	Customer Product Acceptance Form

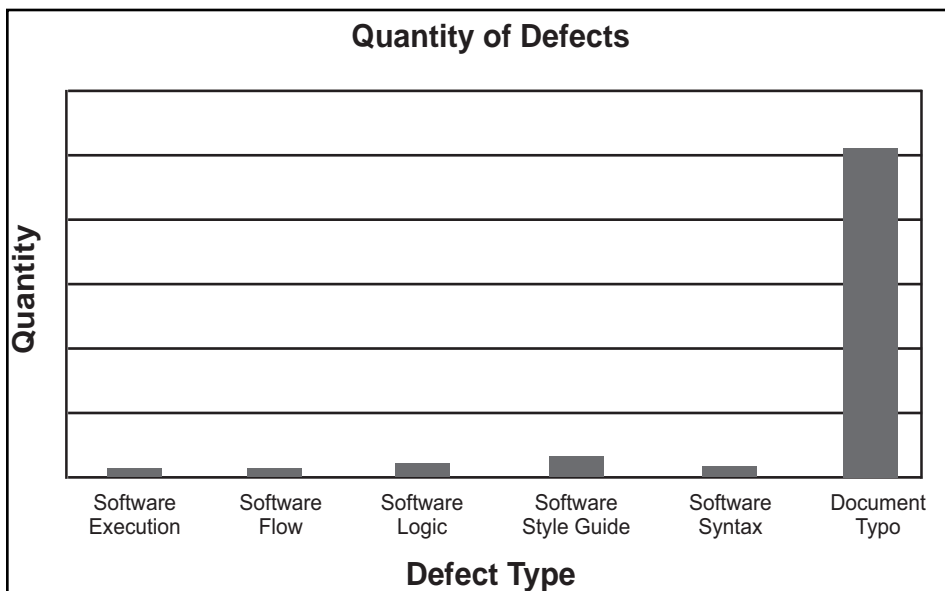Table 1: *Defects Characterized By Category, Type, and Location*

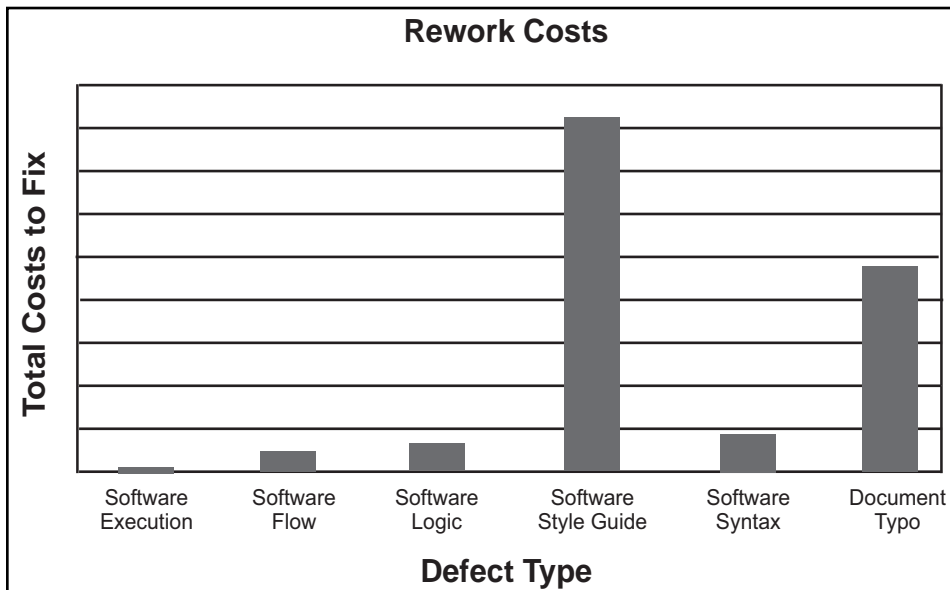Figure 2: *Peel Back - Quantities of Some of the Defect Types*

ilar analysis on the software defects using the same type of metrics developed for documentation defects. Too much information on a chart can make it difficult to understand, so to keep the information presentable, the documentation metrics were displayed on one chart and the software metrics on another. A few items from both categories were selected to display on the chart shown in Figure 2.

The information shown in Figure 2 can be used quite easily to convince a defect prevention team that they need to jump in and begin taking action to reduce the number of typographical errors. But the information presented so far does not answer the question, "Is working the typographical errors the best use of our time?" To answer this, I developed a chart similar to the one shown in Figure 3.

Figure 3 shows an example of the rework costs; this chart was developed to enable an easy comparison between Figures 2 and 3. Presenting and compar-

ing the information in this manner (as shown in Figures 1-3) is a method that you may want to consider to help prioritize your defect prevention activities.

Applying Statistical Process Control

Knowing the information discussed earlier, many teams may think, "We know everything that we need to know. What can statistical process control (SPC) tell us that we don't already know?" To start with, the information shown in Figures 1-3 does not identify whether or not the process is under control, and the charts do not identify random events versus non-random events. Non-random events can be assigned to specific causes, which you may be able to prevent or take into future consideration as a risk.

At least seven *watch-for* indicators have been identified as events that can be assigned to a cause; they have a very low probability of being random in nature. These watch-for indicators include the following:

- One or more points above the upper natural process limit (UNPL) or below the lower natural process limit (LNPL).
- Seven or more consecutive points on one side of the center line.
- Six or more points in a row steadily increasing or decreasing.
- Fourteen points in a row alternating up and down.
- Two out of three consecutive points in the outer third of the control region.
- Fifteen or more points in a row within the center one-third region of the chart.
- Eight or more points on both sides of the control chart with none in the center one-third region of the chart.

Using the same data, I generated the Sample (X) and moving Range (XmR) Control Charts for the total number of defects found during each peer review. The Sample (X) run chart is shown in Figure 4.

The LNPL shown in Figure 4 was not allowed to go below zero because it is impossible to have a negative number of findings. As can be seen in Figure 4, only one anomaly occurred where the number of peer review findings exceeded the UNPL.

I was concerned that by including all defect types in the run chart, I was masking defects that could be assigned to a cause. I then developed individual XmR charts for 18 different types of

Figure 3: *Rework Costs per Defect Type*

defects and for 19 different defect locations (okay, so I need a life). Peeling back the data and looking at the specific defects revealed an additional 18 anomalies where the quantity exceeded the UNPL. Figure 5 shows one of these additional charts, which in this case there were five instances in which the quantity of defects exceeded the UNPL.

The result of this effort identified a total of 19 anomalies[1] in which the quantity of defects exceeded the UNPL. As I started looking at each anomaly, a common attribute appeared in the data. All 19 anomalies pointed back to one small[2] highly skilled team working on a project in which the original proposal was too optimistic and based upon an unproven technology. The project quickly went over schedule as soon as the unproven technology failed to meet or exceed the anticipated productivity. The team was under a lot of pressure from both the customer and management to bring the project back on schedule. The harder the team tried to bring the project back on schedule, the louder the voice of the process became.

As I further analyzed the project's data, I started using this analogy: putting three valves on the end of a garden hose does not increase the flow of the water through the hose. The process capability was limited by constraints within the process such as manpower, equipment availability, and equipment throughput. In essence, the process capability resisted heroic efforts to bring the project back into the contract schedule. When the employees tried to rush through their own personal quality checks, they were met with higher defect rates found during the peer reviews.

SPC Versus Non-SPC
The following is a comparison of the two methods of quantitative analysis.

Non-SPC
The benefit of quantitative non-SPC types of metrics is simplicity. The metrics and charts may seem easier to develop, the metrics may take less time to develop, and the audience may find these charts a lot easier to understand. Depending upon the data collected, these may be about the only metrics the team can develop. One drawback is that you do not necessarily know up front if the causes of the defects are random in nature or attributable to specific causes.

Based upon the software style guide rework costs shown in Figure 3, I rec-
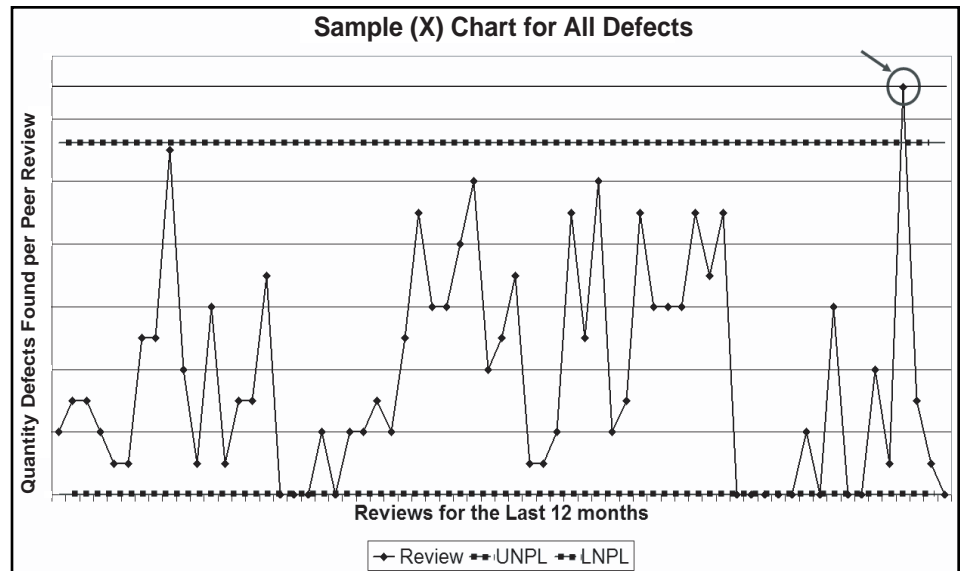

Figure 4: *XmR Sample Run Chart for All Defects*

ommended that the ESEPG first consider a variety of training options to reduce the style guide defects. The corrective actions for these defects could range from creating a heightened awareness (such as a team staff meeting) of the need to follow the style guide, to providing the team with formalized training on it. The cost of implementing each of the proposed solutions can be calculated, the annual rework costs are known, and based upon the perceived success of the proposed solutions, the defect prevention team can determine the appropriate corrective action plan.

SPC
The benefits of applying SPC techniques as a project management tool are that they may help identify problems that could remain hidden by other quantitative analysis methodologies. The cal-

culations are a little more complex, but once you set up your calculations in something like a spreadsheet file then the file can easily be changed for the new set of data.

The results of this analysis led to a decision that every program manager will probably have to make sometime in his or her career. The proper corrective action was obvious, but at first it was not well received by the customer. After determining the process capability, I calculated a new baseline for the project and presented the new baseline to the customer. My analysis included the negative quality impacts experienced from trying to bring the project back on schedule and the argument that the new baseline would reduce life-cycle costs by providing the customer with higher quality products. The damage repair in customer satisfaction took many

Figure 5: *XmR Sample Chart for the Quantity of Defects Found in the Engineering Documentation*
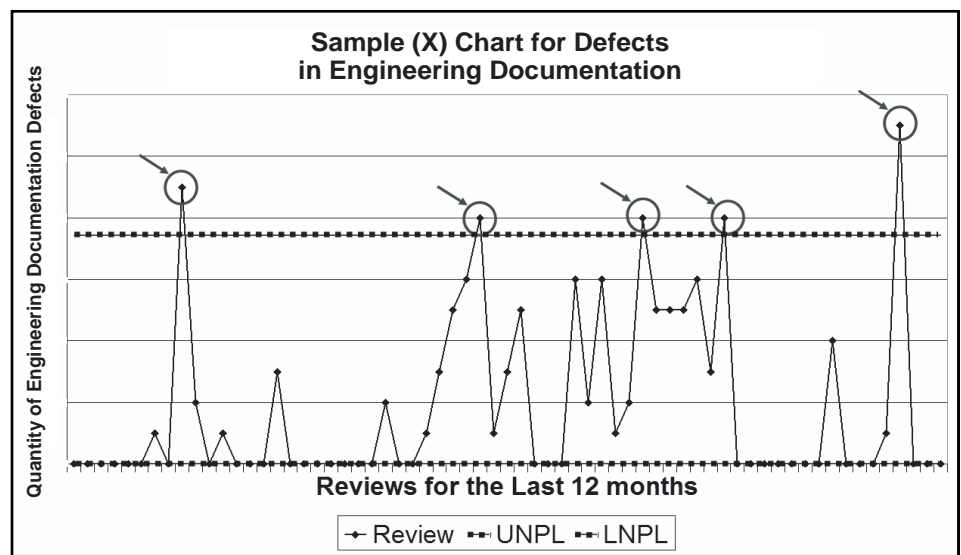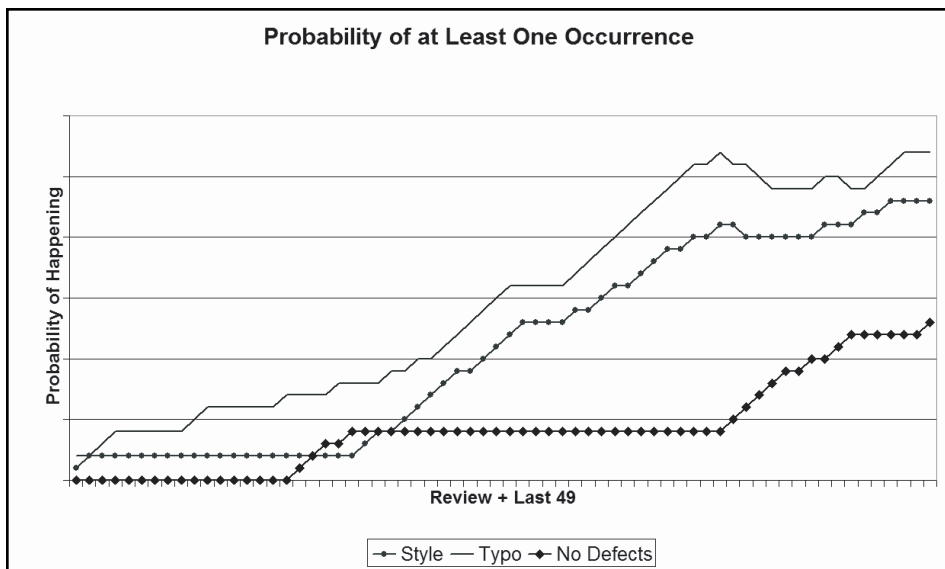
Figure 6: *Moving Window of the Probability of at Least One Defect, or No Defects, Being Found*

months to achieve, but the last feedback that I received was that customer satisfaction did improve over time. The team met the re-baselined plan and provided the customer with a higher quality product.

What Next?
All of the charts discussed in this article provide a historical view of process activities. Displaying the data in a manner that shows trends may enable management to move from reactive management toward proactive management activities. I explored a variety of options for trying to watch for trends in the quality. One option that seemed to give some insight into the process was to show the trend of the probability of the chance of one or more defects being found; for each peer review I set a yes/no flag to indicate whether any defects of that nature occurred. I established the probability calculation based upon the sum of defects found in the last 50 peer reviews. By using the information from the last 50 reviews, I was able to develop a chart with a moving window (last 50) that would show a trend in the data.

I chose to use the last 50 reviews for two reasons. First, it was large enough to give a fair representation of the probability of the defect occurring in the product. The second reason was that even with using a sample size of 50, the time period spanning the reviews was less than a year. Figure 6 shows the trends for two of the defect types; the undesirable trends include the increasing probability of finding style guide and typographical defects. Smaller improvements in other defect types

added up to a noticeable improvement trend in the probability of not finding any defects. The probability of not finding any defects was promising but the undesirable trends again reinforced a need to take action to reduce the style guide and typographical defects.

Conclusion
The three attributes of the product being developed are cost, schedule, and quality. When projects fall behind schedule and/or over-budget, then efforts are made to bring the project back on track, but it is undesirable to do this at the expense of quality. Applying the SPC concepts to the process revealed that our current course of action on one project risked delivering poor-quality products to the customer. In this case, the application of the SPC concepts enabled us to change our course of action to improve the quality of the products delivered to the customer.

As shown earlier, a lot of knowledge can be gained by a careful analysis of the data. By carefully analyzing the data and comparing the perceived benefits versus the costs, the defect prevention teams can select activities that provide the best return on investment.

Final Note
You may find automated charts to be one of your greatest assets, but they can also be one of your greatest liabilities. The person that extracts the data, performs the calculations, and builds the charts seems to have a much better understanding of the data behind the chart than does the person that gets the charts from an automated process.◆

About the Author
David B. Putman for the past year managed system safety, environmental, and engineering data teams at Ogden-Air Logistics Center (OO-ALC) and has recently returned to a technical position in the Software Engineering Division Maintenance Directorate (MAS) at Hill Air Force Base, Utah. Prior to this assignment, he worked in software engineering for 24 years. He has more than 18 years experience in automatic test equipment (ATE), including nine years as a senior engineer in the Avionics Software Support Branch, and more than three years managing ATE workloads. He has also managed F-16 Operational Flight Program System Design and Integration Test teams and the Operational Flight Program Branch. He was the lead of the Software Engineering Process Group when OO-ALC/TIS (prior MAS organization) was assessed to be a Capability Maturity Model® Level 5 organization. He has a bachelor's degree in electrical engineering from the University of Utah and a master's degree in business administration from Utah State University.

Ogden-Air Logistics Center
6054 Dogwood AVE
BLDG 1255
Hill AFB, UT 84056-5816
Phone: (801) 775-2661
E-mail: david.putman@hill.af.mil

Broken Windows

In 1969, Stanford University psychologist Philip Zimbardo conducted an experiment on human nature. He abandoned two similar cars in different neighborhoods – one in the heart of the Bronx, N.Y., the other in an affluent neighborhood in Palo Alto, Calif. He removed the license plates, left the hoods open, and chronicled what happened.

In the Bronx, within 10 minutes of abandonment, people began stealing parts from the alluring car. It took approximately three days to strip the car of all valuable parts. Once stripped of economic value, the car then became a source of entertainment. People smashed windows, ripped upholstery, and chipped the paint – reducing the car to a pile of junk.

In Palo Alto, something quite different happened – nothing. For more than a week, the car sat unmolested. There was no theft, vandalism, or even a scratch. Puzzled, Zimbardo, in plain view of everyone, took a sledgehammer and smashed part of the car. Soon passersby were taking turns with the hammer, delivering blow after satisfying blow. Within a few hours, the vehicle was resting on its roof, demolished.

Among the scholars who took note of Zimbardo's experiment were two criminologists: James Q. Wilson and George Kelling. The experiment spurred their now famous *broken windows* theory of crime. Their premise is that if a broken window remains unrepaired, vandals will soon break a building's remaining windows.

Why is that? Aside from the fact that it is fun to break windows, why does the broken window invite further vandalism? Wilson and Kelling's hypothesis is the broken window sends a signal that no one is in charge, breaking more windows costs nothing, and there are no consequences to breaking more windows.

The broken window is a metaphor for ways behavioral norms break down in a community. If one person scrawls graffiti on the wall, others will soon be spraying paint. If one aggressive panhandler begins working a street block, others will follow. In short, once people begin disregarding norms that keep order in a community, both order and community unravel.

Police in big cities have dramatically reduced crime rates by applying this theory. Rather than concentrating on felonies, they aggressively enforce minor offenses like graffiti, public drinking, panhandling, and littering. This police enforcement sends a signal that broken-window behavior has consequences in a city. If you cannot get away with jumping a turnstile in the subway, you had better not try armed robbery.

At this point, you are wondering what crime in the streets has to do with software development. The broken window theory plays out in software development organizations daily. Software managers inadvertently send signals that no one is in charge and there are no costs or consequences to ignoring project norms. Before you say "not on my project," you might want to look for some classical *broken windows* in your organization.

Problems arise when managers allow prima donnas to dominate, intimidate, and dictate projects. It is tempting to let a technical superstar take the lead, especially for managers who question their own engineering talent, but they will pay in the end. Once ideas are stifled and insults start flying, team members will opt out or limit their contribution to the project. The prima donna will get overloaded and then the vandalism will begin. Broken stained glass is still broken glass. Do you cultivate sages who are inclusive and teach their craft, or prima donnas who hide their weaknesses and feed their insecurities?

Do you have managers whose directions are clear as mud? Like the opaque window in a bathroom, they appear to shed light on the subject but in reality, things are not that bright or clear. After a while, some engineers enjoy these opaque managers because if directions are not clear then accountability is not clear. If accountability is not clear, then this project is a free for all, so start breaking the windows. Are you blocking the light or letting the sunshine in?

Troubles occur when managers exert their authority by hoarding information and tightening control. Collaboration and initiative are dirty words to these comptrollers. Everything runs on maximum management sanction and minimum information sharing. Processes stall or wander, engineers revert to cruise control, and information flows like Molly Brown through a portal window. Do you lead, manage, or choke your projects?

Then there are indecisive managers, the sliding glass doors of management. People are enamored with sliding glass doors until they own one. Then you discover the door is always open when you want it closed and kids are constantly running into it when closed, thinking it is open. Like a sliding glass door, you never seem to be accordant with indecisive managers. They never provide direction and avoid decisions until you make a move, then there they are – blocking progress or letting the air out of your project. Are you indecisive? Need more time to think about it?

Space and time is running out so we will have to discuss the skylight manager, triple-pane glass manager, tinted window manager, two-way mirror manager, and the cockpit canopy manager another time.

The point is, once managers begin disregarding norms that keep order in a project, both order and the project unravel. Repair the broken windows in your management style and order will return.

Amazingly, I think Wilson and Kelling's theory may explain the mystery of software quality. From its first release to present versions, Microsoft Windows was released broken. Distributing broken Windows sends a signal that no one is in charge, there are no consequences, and breaking more Windows software is okay. Software norms break down and our systems vandalized – all from broken windows.

—**Gary Petersen**
Shim Enterprise, Inc.

The Joint Services

S·T·C ►
Systems & Software Technology Conference

18 - 21 April 2005 • Salt Lake City, UT

"Capabilities: Building, Protecting, Deploying"

SSTC reflects the convergence of the Department of Defense's tactical and non-tactical systems, processes, people, and policy in support of our warfighters.

Conference Registration
Opens 2 January 2005

Exhibit Registration
Open now, sign up today!

Be a part of the action today!

www.stc-online.org
800-538-2663

The Joint Services Systems & Software Technology Conference is co-sponsored by:

| United States Army | United States Marine Corps | United States Navy | Department of Navy | United States Air Force | Defense Information Systems Agency |

Software Engineering Division
Ogden Air Logistics Center

Co-Sponsored by
U.S. Air Force
Air Logistics Centers
MAS Software Divisions